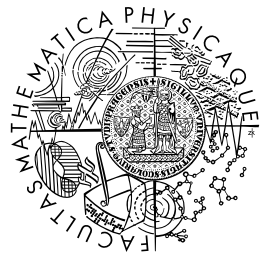


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

DOCTORAL THESIS

Adam Šmelko

Employing Parallel Computing in Data-Intensive Tasks

Department of Distributed and Dependable Systems

Supervisor of the doctoral thesis: Martin Kruliš

Study programme: Computer Science

Study branch: Software Systems

Prague 2024

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I dedicate this thesis to my family, friends, and colleagues who have supported me throughout my studies. I would like to express my gratitude to my supervisor, Martin Kruliš, for his guidance, patience, and valuable feedback.

Title: Employing Parallel Computing in Data-Intensive Tasks

Author: Adam Šmelko

Department: Department of Distributed and Dependable Systems

Supervisor: Martin Kruliš, Department of Distributed and Dependable Systems

Abstract: This thesis studies, develops, and investigates the optimization of data-intensive scientific algorithms using Graphics Processing Units (GPUs) to enhance performance and scalability. The first part of the thesis focuses on the design and implementation of optimized kernels for four key algorithms: hierarchical clustering with Mahalanobis linkage, neighborhood-based dimensionality reduction through EmbedSOM, optimization of cross-correlation algorithms for many small inputs, and stochastic simulation of Boolean networks. In the second part, the thesis builds upon the findings of the first part to propose a Noarr library, which enables the efficient development of high-performance computing (HPC) applications. It emphasizes the critical role of memory optimization in achieving significant performance improvements in HPC and aims to streamline the implementation of these optimizations by providing a novel memory layout and traversal optimization framework. The main contributions of this thesis comprise the implementation of novel GPU optimization techniques, performance improvements of scientific tools of up to three orders of magnitude speedup, advancing data analysis and visualization in bioinformatics and material physics, and the design of new tools for efficient expression of data structure layout and traversal in HPC code. The results of this thesis may be used to enhance the development process of maintainable and efficient HPC applications and guide future research in the field of data-intensive scientific computing.

Keywords: GPGPU, HPC, hierarchical clustering, cross-correlation, memory optimizations, loop transformations

Contents

List of publications

The table below outlines the relative contributions of the authors in each publication, followed by a complete list of references. In the table, an empty circle indicates little to no contribution, a partially filled circle indicates partial to significant contribution, and a full circle signifies work done entirely or almost entirely by the author. Entries marked with an asterisk (*) have not yet successfully passed peer review as of June 2024. The underlined entries represent the main contributions of this thesis.

Relative contributions

Ref.	Channel	Year	Analysis	Implementation	Evaluation	Writing
(a)	Euro-Par	2021	●	●	●	◒
(b)	ICA3PP	2022	◒	○	◒	◒
(c)	arXiv	2022 *	◒	◒	◒	◒
(d)	GPGPU	2024	◒	◒	◒	●
(e)	JPDC	2024 *	○	○	●	◒
(f)	BMC Bioinfo.	2024	●	●	●	●
(g)	PMAM	2024	◒	○	◒	○

Complete list of references

(a) Adam Šmelko et al. “GPU-Accelerated Mahalanobis-Average Hierarchical Clustering Analysis”. In: *European Conference on Parallel Processing*. Springer. 2021, pp. 580–595

(b) Adam Šmelko et al. “GPU-acceleration of neighborhood-based dimensionality reduction algorithm EmbedSOM”. in: *16th Workshop on General Purpose Processing Using GPU*. Association for Computing Machinery, 2024, pp. 13–18

(c) Adam Šmelko et al. “Scalable semi-supervised dimensionality reduction with GPU-accelerated EmbedSOM”. *arXiv preprint arXiv:2201.00701* (2022)

(d) Adam Šmelko et al. “Astute Approach to Handling Memory Layouts of Regular Data Structures”. In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2022, pp. 507–528

(e) *The response for the second round of peer-review process is pending.*

(f) Adam Šmelko et al. “Maboss for HPC environments: implementations of the continuous time Boolean model simulator for large CPU clusters and GPU accelerators”. *BMC bioinformatics* **25** (2024)

(g) Jiří Klepl et al. “Pure C++ Approach to Optimized Parallel Traversal of Regular Data Structures”. In: *Proceedings of the 15th International Workshop on Programming Models and Applications for Multicores and Manycores*. 2024, pp. 42–51

Introduction

General-purpose GPU programming (GPGPU programming) has established a strong position in the domain of high-performance computing (HPC). GPU programming enables the parallel execution of many threads on a single device, exploiting the massive computational power of modern GPUs. These devices are no longer used exclusively for graphics rendering, and since the advent of the CUDA framework (and other frameworks alike), they have become a viable alternative to CPUs for high-performance computing [1]. Nowadays, the top data centers in the world are equipped with thousands of GPUs [2], being used for a wide range of tasks, including machine learning, data analytics, and scientific computing [3–5].

Naturally, GPUs are not a silver bullet as they are not suitable for any task. GPU programming poses many challenges, including complex hardware architectures, heterogeneous programming models, memory management, performance tuning, and portability. The ever-growing list of new CUDA features, such as dynamic parallelism [6], asynchronous data copy [7], independent memory pools [8], kernel launch buffering [9], distributed shared memory [10], tensor cores [11], and many more, is a double-edged sword; it aids programmers in achieving peak GPU performance in their applications but at the cost of increased programming complexity and an expert architecture knowledge requirement. Generally, GPGPU programming is an iterative process of developing a solution, profiling it, and optimizing the bottlenecks. Apart from that, there is generally *no single optimal solution* to the problem. The programmers must develop multiple variants that must be empirically tested to find the most suitable optimization for a specific set of input parameters [12].

This thesis contributes to addressing the challenges of GPGPU programming in two interconnected topics: Firstly, the thesis couples a set of data-intensive scientific algorithms and their GPU optimizations; each contains a detailed discussion of their concurrency opportunities, memory operation analyses, and the choice of the most suitable optimization variant. Each work employs a slightly different tool in the GPU programming toolbox. In the second part of the thesis, we introduce the Noarr library, which builds upon the findings and expertise

gained from the results of the previous part. We identified that the primary factor of many optimizations is the order of accessing data in the memory, generally referred to as *memory optimizations*. Noarr adds memory layouting primitives and provides a way to compose and traverse them in a customizable manner. The main goal of the library is to aid in writing more expressive, maintainable, and modular HPC code.

Structure of the thesis The thesis is separated into three chapters. Chapter 1 provides an introduction to GPU programming and highlights the most significant performance-related pitfalls. In Chapter 2, we focus on the GPU optimizations of four scientific algorithms. Section 2.1 describes the first scientific algorithm, the Hierarchical Clustering algorithm with Mahalanobis-based distance, and details the ways of overcoming problems of imbalanced workloads. In Section 2.2, EmbedSOM, a dimensionality reduction algorithm, is discussed from the perspective of caching and utilization of multiple memory hierarchies. Further, we detail the importance of keeping high arithmetic intensity when implementing a cross-correlation algorithm in Section 2.3. We conclude the chapter with Section 2.4, which describes MaBoSS, a Monte Carlo simulator of biological systems, and the runtime compilation capabilities of GPUs. Chapter 3 introduces the Noarr library. The chapter is logically divided into two subparts: the techniques of expressing memory layouts as first-class citizens (Section 3.1) and the ways of traversing them in a customizable manner (Section 3.2). Appendix A contains the peer-reviewed publications which support the contributions of this work.

Chapter 1

Introduction to GPU programming

Graphics Processing Units (GPUs) have evolved from specialized hardware designed solely for rendering graphics to powerful, general-purpose processors capable of handling complex computations. In the early years, 3D acceleration carts, as they used to be called, were devoted to offloading the rendering tasks, such as drawing pixels, filling polygons with textures, or computing geometry, from the CPU [13]. And because processing pixels is inherently parallel, GPU architectures were being developed with different paradigms compared to ordinary CPUs. GPU chip semantics is oriented on high performance achieved by immense parallelism, with a design that promotes further scalability. Its die surface is largely occupied by compute cores with very little space dedicated to cache and control, which contradicts the CPU design (the graphical reference depicted in Figure 1). Thus, GPUs can be much more powerful than CPUs if the solution is carefully designed for it [14].

The shift from specialized graphics processing to general-purpose computing was marked by the introduction of programming frameworks such as CUDA (Compute Unified Device Architecture) by NVIDIA and OpenCL (Open Computing Language) by Khronos group. As extensions to C/C++, these frameworks simplified the process of writing code for GPU and streamlined the adoption of GPU for general-purpose programming [15]. CUDA, in particular, has become the de facto standard for GPU programming due to its ease of use, performance, and wide adoption in the scientific community.

In this chapter, we provide an overview of the GPU hardware architecture and the CUDA programming model. We also discuss the main optimization techniques used in GPU programming, which are essential for achieving high performance on GPUs. The knowledge presented in this chapter serves as a foundation

for the subsequent chapters, where we detail the GPU optimizations of scientific algorithms and describe the parallelization use-cases of the Noarr library.

1.1 GPU Hardware Architecture and Programming Model

Generally, a programming model serves as a high-level abstraction of the hardware architecture, allowing programmers to write code that can run and scale well without prior knowledge of the underlying hardware specifics. The terminology within the model varies depending on the programming framework or GPU vendor, but the core concepts remain the same. To maintain conciseness, we continue with the description in CUDA terms.

Let us start with the hardware. The main building block of a GPU is the *Streaming Multiprocessor* (SM). The diagram in Figure 2 displays its composition: SM couples hundreds of compute cores, a register file, L1 cache also called *shared memory*, and schedulers responsible for independent scheduling of groups of threads. There are multiple SMs in a GPU, their number highly correlating with a GPU cost — the more high-end the GPU, the more SMs it holds. All SMs are connected via a high-speed interconnect to the GPU RAM, the *global memory*.

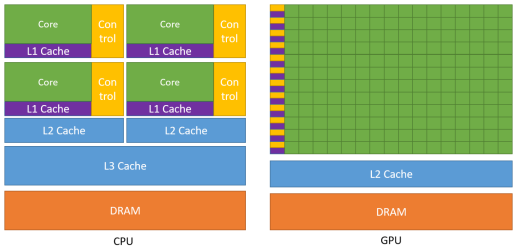


Figure 1 Comparison of CPU and GPU chip design [16].

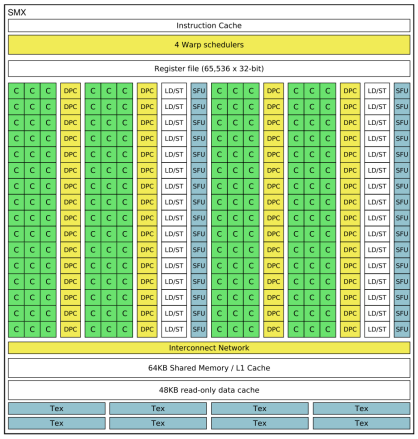
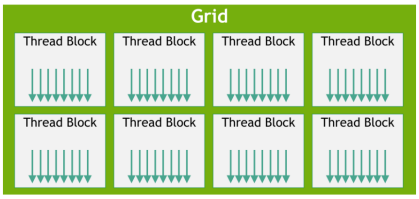
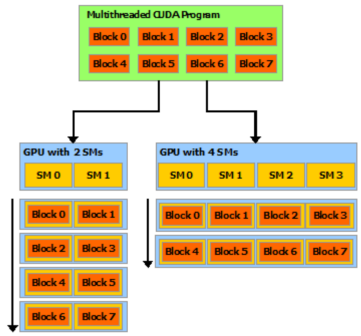


Figure 2 Diagram of a streaming multiprocessor [14].

The abstraction of this highly parallel architecture is based on organizing threads into hierarchical levels. The basic units of execution are *threads*. These are organized into *blocks*, groups of threads *executed on the same SM*, sharing a register file and the shared memory. Lastly, a group of blocks that is executed independently on multiple SMs is called a *grid*. Thus, the grid, as depicted in



(a) GPU grid of 8 blocks organized in 2×4 fashion, each block consisting of 8 threads.



(b) The assignment of blocks in a grid to SMs on GPUs with various SM counts.

Figure 3 A diagram of a grid ands its SMs assignment [16].

Figure 3a, defines the amount (and dimensionality) of concurrent work to be done on a GPU. Figure 3b shows how the hierarchy of threads maps to GPU hardware, abstracting away the details of the hardware architecture in the process.

Conceptually, this programming model is based on *Single Instruction, Multiple Data* (SIMD) architecture, which is a type of parallel processing where a single instruction is executed on multiple data streams. In practice, SM issues threads in groups of 32, called *warps*, which execute in *lockstep* — all performing the same instruction at once. A more accurate description of this type of processing is *Single Instruction, Multiple Threads* (SIMT), which is nowadays used to describe the GPU programming model.

As a consequence of SIMT architecture and lockstep execution, GPUs favor *data-parallel problems*. These are problems such that their solution requires applying the same function to its whole domain (e.g., an n-dimensional array). The data-parallel problems are well-suited for GPUs, as they allow all threads to execute *the same instruction* but on *different data* [14].

1.2 Optimizing for GPU performance

Contemporary GPU devices are becoming increasingly sophisticated and are equipped with a variety of control-flow mechanisms, interthread communication, atomic instruction, and others, which allow for a broader range of problems to be solved on GPUs. However, applying these features correctly is not trivial, as it requires a deep understanding of the architecture and programming model. Thus, achieving optimal performance on a GPU requires careful consideration of various aspects. Let us detail the most important ones to watch for when optimizing GPU code [13].

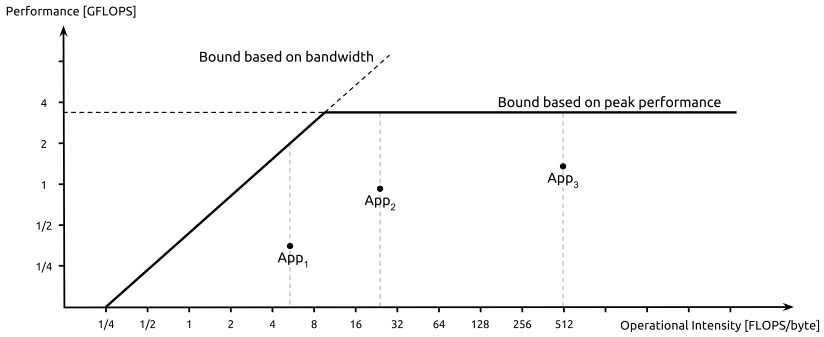


Figure 4 The example of a roofline diagram [18]. The x axis denotes arithmetic intensity (also called operational intensity) and y axis shows attainable performance.

1.2.1 Arithmetic intensity

The peak performance of contemporary high-end GPUs is nowadays measured in TFLOPS (trillions of floating-point operations per second). In order to determine an actual achievable performance of a program, memory bandwidth needs to be considered virtually always. Comparing peak performance and memory bandwidth of current GPUs, we quickly conclude that the cores can not be fed data at the same rate as they can compute.

Therefore, one of the most important constants in GPU programming is *ops-per-byte* ratio. It is computed as the ratio of math and memory bandwidth, and it simply determines how many math operations per transferred byte must be executed to achieve the theoretical maximum performance. For example, NVIDIA V100 GPU achieves 14 TFLOPS peak single-precision performance with a memory bandwidth of 900 GB/s. This results $\frac{14}{0.9} \approx 15$ ops-per-byte ratio. As a simple rule of thumb, a program that performs less than 60 single-precision floating point operations per single 4-byte load/store will be *memory-bound* and a program that performs more operations will be *compute-bound*. Such a property of a program is called *arithmetic intensity* and is a key factor in determining the performance of a GPU kernel.

To get more detail in hardware utilization than just comparing ops-per-byte ratio with arithmetic intensity, modern profilers include the *roofline model* [17]. It is a graphical representation of a hardware attainable performance, given the algorithm arithmetic intensity. Figure 4 shows an example of a roofline diagram, where the x and y axes represent the arithmetic intensity and the performance, respectively. The horizontal lines represent the achievable performance of single- and double-precision floating-point operations and the diagonal line represents the memory bandwidth. The roofline model can be used to determine the performance bottlenecks of a program and to guide optimization efforts.

Strictly speaking, there are two ways how to decrease the gap between arithmetic intensity and ops-per-byte ratio: If a program allows, its operation may be reordered such that fetching the same data multiple times is avoided, promoting *data reuse*, which in turn increases the arithmetic intensity. Another way is to decrease the ops-per-byte ratio by utilizing the GPU memory hierarchy, which we discuss in the following paragraphs.

1.2.2 Memory hierarchy

A GPU thread hierarchy is accompanied by *memory hierarchy*. As alluded in the previous paragraph, global off-chip memory has the highest latency and the lowest bandwidth. To avoid thread stalls measured in hundreds of cycles due to fetching data from global memory, other levels of memory hierarchies should be utilized. This process is generally called *caching* but can be also interpreted as *data sharing*, as the data is shared among threads in the same thread hierarchy.

Depending on the desired level of data sharing within the thread hierarchy, programmers can use various types of memory (as depicted in Figure 5). On a block level, on-chip shared memory can be accessed with a magnitude lower latency and higher bandwidth than global memory. In extreme data sharing cases, threads in the same warp can use the register file as a cache with close-to-zero zero latency. However, the higher the proximity of the specific memory type to the compute cores, the lower capacity it carries. Contemporary GPUs encompass 64k of 32B registers, $100 - 200$ kB of shared memory, and $8 - 16$ GB of global memory for consumer-grade GPUs and up to 80 GB for the highest-end datacenter-grade GPUs.

To sum up, the key to achieving high performance on a GPU is to minimize global memory accesses by smartly dividing the work in a thread hierarchy such that faster memory levels are utilized. Moreover, since the ops-per-byte ratio is inversely proportional to the memory bandwidth, a program using shared memory or registers as caches does not carry such significant requirements on arithmetic intensity to achieve maximum performance.

1.2.3 Memory coalescing

Fetching data from global memory is serviced in up to 128B wide transactions. If threads in a warp request a single byte, the GPU will fetch the whole transaction chunk, wasting the memory bandwidth on data that will never be used and negatively impacting the ops-per-byte ratio. To utilize the memory bandwidth fully, the aggregated memory accesses in a lockstep should coalesce into continuous memory locations. Figure 6 illustrates this concept.

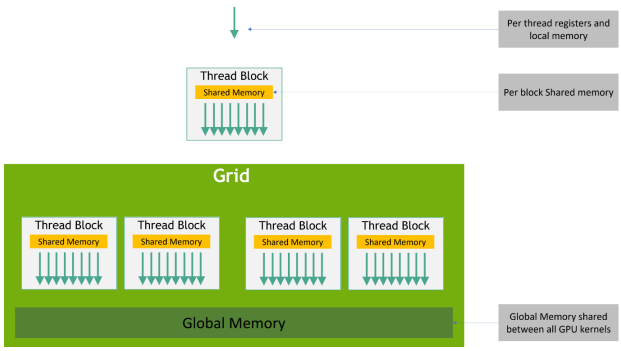


Figure 5 A thread hierarchy of GPU (left) and its corresponding memory hierarchy (right) [16].

In practice, whether memory is transferred in an optimal number of transactions is directly influenced by the layout of the data structures in use. One example may include using a column-major layout for the right-hand side matrix in matrix multiplication or avoiding the usage of linked lists.

1.2.4 CPU–GPU data transfers

Transferring data between CPU and GPU memory spaces can pose a significant bottleneck when developing GPU programs. Usually, the interconnect which is used to transfer data between the CPU and the GPU has a much lower bandwidth even than the GPU global memory. Therefore, it is essential to avoid redundant data transfers between the host and the device.

In true data-parallel algorithms, this is generally not a big issue because these data transfers can be overlapped with computation. Such overlapping is possible by GPU *task queues*, which allow the execution of multiple programs as well as CPU–GPU memory transfers in parallel. This hardware feature is exposed by the software abstraction called *streams*. Figure 7 show the ideal scenario, where one would use multiple streams to overlap three main operations always

Figure 6 Warp accesses coalesced into 4 memory transactions [16].

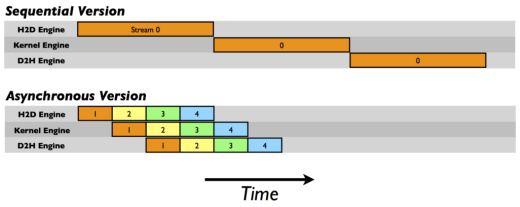


Figure 7 Time diagram of non-overlapped (top) and overlapped (bottom) transfer and computation [19].

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

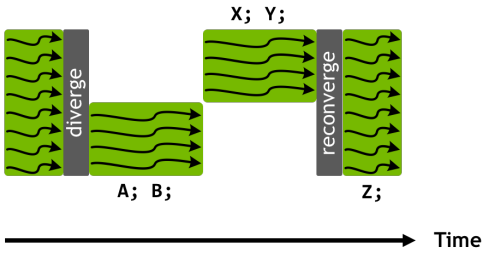


Figure 8 An example of diverging code with a diagram of how warp threads would be scheduled [20].

present when programming on GPUs: transferring input from CPU to GPU, GPU computation, and transferring output back from GPU to CPU.

1.2.5 Thread divergence

The downside of the highly parallel SIMT architecture is handling diverging execution paths in the code. If threads in a warp take different `if` or their iteration count over a `while` loop does not match, the execution of these diverging paths will be serialized, as depicted in Figure 8.

Intra-warp branching should be avoided to keep the number of active threads high for the longest amount of time. Divergence can be mitigated by lifting the branching to the inter-warp level, replacing branches with arithmetic operations if possible, or unrolling warp-wise loops.

1.2.6 Thread occupancy

The GPU is designed to run thousands of threads in parallel and tens of thousands of threads concurrently. Increasing the number of concurrent threads above the limit of compute cores is crucial for GPU utilization as it allows *hiding memory latencies* by switching between active threads. A single SM can accommodate thousands of concurrent threads, although the number of accommodable cores is a magnitude lower. The SM takes advantage of this high discrepancy to aggressively context switch between different warps when the currently scheduled ones are waiting for a latency. Since all the thread contexts reside on-chip, the context switch has negligible overhead, and, provided there are enough available concurrent threads, it can effectively hide the memory latencies.

The number of concurrent threads per SM is called the *thread occupancy*. It denotes the ratio of active threads to the maximum number of threads that can be executed concurrently on an SM. This metric is influenced by the resource requirements of a program: Increasing the shared memory usage per block or reg-

ister usage per thread decreases the thread occupancy, which in turn decreases the parallel capacity of the GPU device.

However, a GPU code that achieves high thread occupancy uses less on-chip resources, so it is required to use slower global memory more often, which in turn decreases the computational efficiency of a thread. Thus, thread occupancy and thread efficiency compete with each other, and their optimal ratio for a given hardware should be finely tuned to achieve the best performance.

1.3 Summary

These are a few of the most dominant optimization techniques used in GPU programming. For the past few years, GPU hardware has evolved extraordinarily rapidly to satisfy the demand for hardware that can train artificial intelligence models with billions of parameters. This has brought many new features, making the device more versatile and powerful. Let it be more advanced SMs, independent thread scheduling, *block cluster*, which brings a new element in the thread and memory hierarchy, specialized matrix multiplication-optimized *tensor cores*, which increase ops-per-byte ratio ten-fold, or *tensor memory access*, which allow asynchronous copying of non-sequential data efficiently, ...

In summary, the field of GPU programming is vast, and optimization techniques are often intertwined and compete with each other. The optimal solution is usually a compromise between the techniques, which is highly dependent on the specific properties of the problem. In the following chapter, we will describe our work of porting scientific algorithms to GPUs using the terminology and knowledge established in this section as guidelines to detail the main optimization challenges for each of them.

Chapter 2

Employing GPUs in scientific algorithms

The information age has brought a massive increase in the amount of data that is being collected and processed. The *data explosion* can be observed in virtually every field of computer science. In the scientific domain, this phenomenon is multiplied by increasingly powerful data acquisition devices such as cytometers in bioinformatics, seismometers in geology, and others [21, 22]. The amount of data is too large to be processed in the required quantum of time, and in order to alleviate this apparent pressure, the corresponding algorithm (which typically has super-linear time complexity) must be ported to massively parallel architectures, such as GPUs. The provided advantage can result in the ability to assess massive amounts of data, use more detailed processing methods, or analyze the data in real time [13, 14].

This chapter summarizes the process of porting scientific algorithms to GPUs and highlights the challenges of optimizing for GPU architectures.

2.1 Hierarchical clustering with the Mahalanobis linkage

In the field of bioinformatics, hierarchical clustering is a popular method for analyzing various types of data. In general, hierarchical clustering is an unsupervised machine learning method that aims to group the data points into clusters according to some *linkage criterion*. Clustering is an iterative method that begins with each data point interpreted as a single cluster and, in each iteration, the two most similar clusters are merged with respect to a linkage criterion until a single cluster remains. There are many cluster linkage criteria used in the field, each with advantages and disadvantages. Perhaps the most common linkage may

be the *centroid linkage*, which defines cluster similarity as the distance between their centroids (the mean points). In the domain of bioinformatics, hierarchical clustering with the *Mahalanobis linkage* is used, which is a more sophisticated method suitable for analyzing multidimensional single-cell cytometry datasets.

2.1.1 Background

Mahalanobis hierarchical clustering was proposed by Fišer et al. [23] as a valuable tool for the analysis of flow cytometry datasets. Due to the ever-increasing rate of data acquisition, the size of flow cytometry datasets is constantly growing. Modern flow cytometers are able to process samples with millions of cells and measure tens of parameters (point dimensions) simultaneously. More traditional methods, such as manual gating, are not suitable for such high-dimensional data and are heavily observer-dependent. Automated hierarchical clustering methods offer a viable solution but often fail to reflect the elliptical shapes of flow cytometric populations.

For this purpose, the authors proposed the Mahalanobis linkage. This allows for a very natural formation of clusters in biological data. However, the benefits come at a cost of high time and space complexity. The authors claim the upper usable bound of dataset size to be 10^4 using the original `C` application. They state that the quadratic space complexity makes for the most significant limiting factor, leaving it unfeasible to analyze bigger datasets on current desktop computers.

The authors tried to alleviate this limitation by implementing the approximation of *apriori clusters*, where the data was pre-clustered using a naïve Euclidean-based linkage, reducing the overall size of the dataset. This increased the usability of the original code to datasets of around 10^6 data points within an interactive environment [24]. Still, the proposed optimization could not keep up with the contemporary data acquisition methods, generating datasets of millions of multidimensional data points.

2.1.2 Algorithm complexity

Mahalanobis linkage uses *Mahalanobis distance* [25] to measure the similarity between clusters:

Definition 1 (Mahalanobis distance)**.** *Suppose a probability distribution C on $\mathbb{R}^d$ with mean $\bar{C} \in \mathbb{R}^d$ and a covariance matrix $\boldsymbol{cov}(C)$. If the matrix $\boldsymbol{cov}(C)$ is regular, we define the* Mahalanobis distance *between $u \in \mathbb{R}^d$ and C as*

$$d_{Maha}(u, C) = \sqrt{(u - \bar{C})^T \boldsymbol{cov}(C)^{-1} (u - \bar{C})}. \tag{2.1}$$

If we generalize a centroid of a cluster to a probability distribution, Definition 1 can be used to define the Mahalanobis distance between a point and a cluster. To extend the definition to a distance between two clusters, we use the following equation [23]:

$$\delta_{\text{Maha}}(P, Q) = \frac{d_{\text{Maha}}(\bar{P}, Q) + d_{\text{Maha}}(\bar{Q}, P)}{2} \tag{2.2}$$

To illustrate the measure of the Mahalanobis distance, let us suppose we have two elliptic clusters. In the means of proximity, the measure favors such clusters that their ellipses are alongside rather than in a prolongation of one another [26]. Only when the objects of a cluster form a spherical shape, this dissimilarity measure is proportional to the Euclidean distance with a corresponding linkage (as depicted in Figure 9).

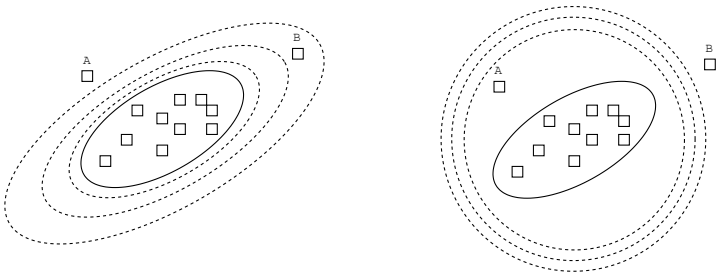


Figure 9 The comparison of the Mahalanobis (left) and Euclidean (right) distance in the context of elliptic clusters. The contour lines represent the space of the equal distance from the cluster centroid.

Finally, Algorithm 1 summarizes the Hierarchical Clustering (HC) with Mahalanobis linkage. Line 2 initiates $n-1$ iterations, each one starting with one less cluster to merge. Denoting d as the dimension of the data, the time complexity of the Mahalanobis distance computation is $\mathcal{O}(d^2)$, according to the Equation 1 and the fact that the size of a covariance matrix is $d \times d$. The time complexity of the covariance matrix computation is $\mathcal{O}(d^2 \cdot c)$, where c is the size of a cluster, and its inverse is computed in $\mathcal{O}(d^3)$.

The standard HC, which utilizes a dissimilarity matrix, has a time complexity of $\mathcal{O}(n^3)$ and space complexity of $\mathcal{O}(n^2)$. There are other variants of HC which influence the closest cluster pair retrieval and the amount of distances to recompute after each iteration. Other variants include HC with the nearest neighbor

Algorithm 1 Mahalanobis Hierarchical Clustering Analysis

1: **procedure** MHCA(Set of clusters $C = \{c_1, \dots, c_n\}$, dimension $d \in \mathbb{N}$)
2: **while** $|C| > 1$ **do**
3: **for** all cluster pairs (c_i, c_j) **do**
4: Compute $\delta_{\text{Maha}}(c_i, c_j)$ ▷ Time: $\mathcal{O}(d^2)$
5: **end for**
6: Find the closest pair of clusters (a, b)
7: Update C by merging a and b into c
8: Compute $\mathbf{cov}^{-1}(c)$ ▷ Time: $\mathcal{O}(d^2 \cdot |c| + d^3)$
9: **end while**
10: **end procedure**

array, which trades the linear memory complexity for a worse time complexity on average, or the HC with priority queues, which reduces the time complexity to $\mathcal{O}(n^2 \log n)$ for the sake of bigger memory requirements [27].

2.1.3 Implementation

From the perspective of GPU programming, Mahalanobis HC is a challenging problem. Algorithms with high memory requirements are generally unfavorable for GPUs due to their relatively limited size of global memory (refers to Memory hierarchy in Section 1.2). There has been a lot of work on overcoming this limitation, both from NVIDIA and the scientific community [28–30]. CUDA has introduced *unified memory* [16], which allowed GPUs to work on data that exceeds GPU memory capacity by seamlessly copying data from CPU to GPU on page fault. This optimization can help only to a certain extent, as it can only scale to the size of CPU RAM and the data are usually sent through high-latency small-throughput interconnect (refers to CPU–GPU data transfers in Section 1.2). Therefore, in our work, we experimented with other well-known HC variants that trade higher time complexity for a linear memory complexity — *HC with the nearest neighbor array* [27].

The other obstacle of Mahalanobis HC is the greatly imbalanced workload. In the first iterations, the runtime is dominated by the complex Mahalanobis linkage computation over many pairs of n clusters. But as the number of clusters decreases, the hot spot becomes the computation of the covariance matrix of a merged cluster (Algorithm 2, Line 8). As a result, for the most time during the computation the GPU is not fully utilized either due to small amount of parallelism in covariance matrix computation or in the similarity computation. For that case, we designed a workload using CUDA *streams* (refers to CPU–GPU data transfers in Section 1.2), which enabled running these tasks in parallel. Conse-

quently, this allowed us to keep the utilization of more GPU cores during the whole runtime of the algorithm.

2.1.4 Results

The optimized Mahalanobis HC achieves a speedup of over $1400\times$ compared to the original serial CPU implementation. We benchmarked the application on the real-world single-cytometry datasets. The biggest dataset which we obtained, the Samusik-All [31] (841 thousand 39D points), was able to finish in the order of minutes compared to the order of days. Furthermore, the application has been distributed as a R package `gmhc` to fit workflows carried out by a bioinformatics community. To the best of our knowledge, the package enabled the scientists to analyze big datasets as a whole without the apriori clustering approximation, increasing the accuracy of the analyzed data and decreasing the turnaround time of the analysis.

2.2 Neighborhood-based dimensionality reduction

Complementary to hierarchical clustering, a different approach to displaying cytometry datasets is *dimensionality reduction* (also called *embedding*), in which multidimensional cells are projected into a 2-dimensional plane, a picture, which shows cells arranged in groups with common properties. This methodology allows for a fast and reliable way to analyze cell populations, their relative size, and the presence of various features. The currently used dimensionality reduction tools are typically based on the principle of optimizing a low-dimensional embedding while preserving high-dimensional properties of interest. However, the most popular tools following this methodology, such as t-SNE [32], UMAP [33] or TriMAP [34], can suffer poor performance on large data due to the need to examine a nontrivial subset of $\binom{n}{2}$ relations (such as the pairwise distances) between n data points [35].

A lot of effort has been dedicated to optimizing the performance of these algorithms. Solely for t-SNE, we can find multiple works related to this topic [36–39]. Regardless of these developments, the processing time of these algorithms scales super-linearly with the number of data points, which inevitably leads to the need of data downsampling. *EmbedSOM* algorithm, introduced by Kratochvíl et al. [35], is designed to overcome this limitation. The costly parts of the previous methods can be omitted by creating a smaller model of the data obtained (not exclusively) by *self-organizing maps*. EmbedSOM uses the information of such an

Algorithm 2 EmbedSOM

procedure EMBEDSOM($X \in \mathbb{R}^{n \times d}$, $L \in \mathbb{R}^{g \times d}$, $l \in \mathbb{R}^{g \times 2}$, $k \in \mathbb{N}$)
  for $i \in \{1 \dots n\}$ **do** ▷ For each high-dimensional point
    Find k nearest landmarks from L
    Score the k nearest landmarks according to the distance to X_i with $s_1, \dots, s_k$
    for $(u, v) \in \{1, \dots, k\}^2$ **do** ▷ For each pair of nearest landmarks
      Compute $D_{uv}(X_i)$ by projecting X_i orthogonally onto a line between L_u and L_v; We define $d_{uv}(x)$ similarly for l_u and l_v
    end for
    Find x_i, such that $\sum_{u,v} s_u \cdot s_v \cdot (D_{uv}(X_i) - d_{uv}(x_i))$ is minimized
  end for
end procedure

approximated manifold to compute the final embedding, retaining a competitive quality of the visualization.

Algorithm 2 shows the overview of EmbedSOM. It assumes a set of n high-dimensional points $X \in \mathbb{R}^{n \times d}$ and the smaller data model: a set of $g << n$ high-dimensional landmarks $L \in \mathbb{R}^{g \times d}$, and a set of g low-dimensional landmarks $l \in \mathbb{R}^{g \times 2}$. For each input point, $k < g$ nearest landmarks from L are found and assigned scores according to their distance. Finally, we compute the embedding such that the difference between distances from l landmarks and the embedded point and L landmarks and the original point is minimized. The minimization problem is reducible to a linear system of equations with two variables.

With such a description of the algorithm, we can deduce the time complexity for a single data point processing: The first line of the for loop is the well-known k-NN problem, with the optimal time complexity of $\mathcal{O}(d \cdot g \cdot \log k)$. The remainder, which we may call the embedding step, has a time complexity of $\mathcal{O}(d \cdot k^2)$. Since it holds that $k < g << n$, EmbedSOM achieves sufficient scaling; the embedding of 24 million cells with 36 markers can finish under an hour on common hardware, compared to around 2 days using UMAP [40]. Figure 10 visualizes the EmbedSOM process for a single data point x.

2.2.1 Interactive opportunities of GPU implementation

Although the EmbedSOM CPU implementation already provided good performance, it still had an untapped parallelization potential and a potential for enablement as an interactive method in cytometry data analysis. Therefore, we followed up on it in our work.

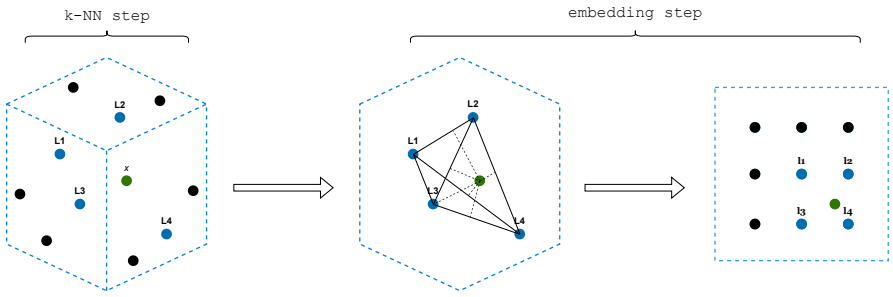


Figure 10 The visualization of the EmbedSOM algorithm, with k-NN and embedding steps highlighted.

The main challenge of EmbedSOM is low *arithmetic intensity* (refers to Arithmetic intensity in Section 1.2). Note that although the arithmetic intensity is primarily determined by the algorithm, it can be influenced by its implementation in a major way. For GPUs, there are many ways to fight the arithmetic intensity of an algorithm, such as kernel fusion [41], leveraging the memory hierarchy [42] or by reordering data accesses [43] and optimizing data transactions [44]. Generally, these approaches can be distilled into two groups: *Data sharing* and *latency hiding* (refers to Memory hierarchy and Thread occupancy in Section 1.2).

We experimented with both approaches in our EmbedSOM implementation. Although L2 caches can partially handle data sharing, GPUs typically hardly benefit from them due to their low cache-to-core ratio [16]. Therefore, as one variant of k-NN part, we used *shared memory*, a programmable cache, to store the data and landmarks. These were then used to compute all the possible pairwise distances before loading the next batch to maximize the arithmetic intensity. The second variant of the k-NN part used a modified *bitonic sort* to find the top-k landmarks. This approach has the hidden benefit of low per-thread resource requirements, resulting in higher maximum GPU occupancy and, consequently, better latency hiding (refers to Thread occupancy in Section 1.2).

The embedding part of EmbedSOM does not offer such caching opportunities, as each point has a different set of nearest landmarks. The only reuse can happen on the landmarks themselves. And since their count can be limited in some parameter configurations, we experimented with techniques that increase the memory bandwidth, such as *vector load instructions*.

2.2.2 Results

After the thorough benchmarking process, we selected the most performing variants of both steps and combined them into a complete implementation of the

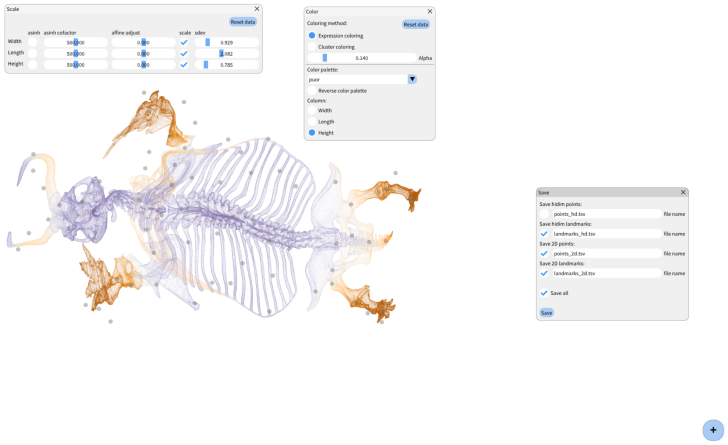


Figure 11 Mammoth skeleton dataset visualized in BlosSOM.

EmbedSOM algorithm. We increased the performance by $200 - 1000\times$ over the serial CPU version and $3 - 10\times$ over a naïve GPU implementation. Furthermore, the achieved speedup enabled the interactive data visualization and was integrated into the graphical application *BlosSOM* [45] (a screenshot included in Figure 11). Thanks to the added performance, the application can project datasets of up to a million points with a maintained frame rate above 30 frames per second on consumer hardware. This contribution pushes the boundary of EmbedSOM into a semi-supervised dimensionality reduction domain, allowing users to effectively and intuitively visualize data with real-time feedback.

2.3 Cross-correlation optimized for small inputs

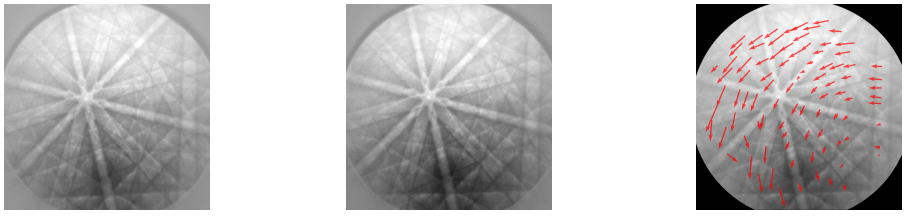
Leaving the realm of cytometry data analysis, we will focus on a highly data-bound problem: cross-correlation. This algorithm is a cornerstone of many scientific fields, such as image processing, seismology, material physics, and, with the advent of convolutional neural networks, machine learning. As one interpretation, it computes a similarity of two data series obtained by sliding one over the other and computing the dot product of the overlapping parts. The output is typically post-processed to get richer information, such as Earth's underground structure in seismology [22], road lane detection in computer vision [46], or acoustic location in signal processing [47].

The problem of cross-correlation performance was initially introduced to us by the physicists from the Department of Physics of Material at Charles University in Prague. They used the algorithm to analyze the diffraction pattern of metallic alloys from electron backscatter diffraction (EBSD) cameras to obtain

the material *deformation*. That can be used to study the material characteristics, such as elastic strain or lattice rotation.

A typical input consists of a *reference* 2D grayscale image of the analyzed material and multiple images of the *deformed* material (as depicted in Figure 12). In the $1 : N$ relation, the reference image is cross-correlated with each deformed image to find the most similar shift, which suggests the direction of deformation. However, different parts of the image may be deformed differently. Therefore, images are typically divided into smaller parts, and the cross-correlation is computed for each part separately, requiring multiple $1 : N$ cross-correlations. Considering such input configuration, the computation can quickly get expensive, and the analysis becomes performance-constrained.

The reference Python script provides the physicists with the computational throughput of tens of patterns per second. However, modern EBSD cameras produce data at a magnitude higher rate [48]. Nevertheless, thanks to the parallel nature of the problem, this is a perfect candidate for GPU acceleration.



(a) Reference pattern **(b)** Deformed pattern **(c)** Deformation visualization

Figure 12 A visualization of FeAl alloy deformation computed using cross-correlation

2.3.1 Implementation challenges

The cross-correlation of functions $f, g : \mathbb{C} \to \mathbb{R}$ is defined as

$$(f \star g)(\tau) = \sum_{i=-\infty}^{\infty} f(i)g(i+\tau). \tag{2.3}$$

Figure 13 depicts the visual representation of the equation applied on a discrete case of two 2×2 matrices. The matrices are shifted to produce all the possible overlaps (Figure 13, right). For a single shift, only overlapping elements contribute to the computation: overlapped pairs are multiplied and summed together into a single output matrix element (Figure 13, left). From the implementation perspective, a trivial solution can be developed rather quickly by defining four nested loops, two for the shift and two for the overlap computation as shown in Listing 1.

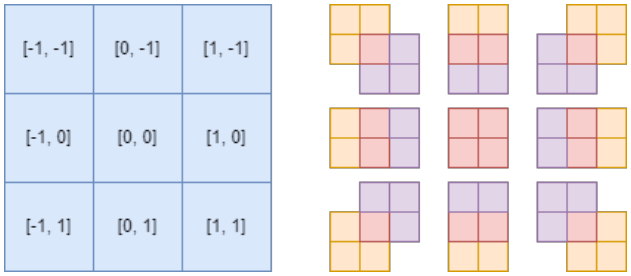


Figure 13 Visual representation of the cross-correlation. Yellow and purple matrices correspond to the input matrices, and the blue matrix depicts the cross-correlation output. The coordinates on the blue matrix correspond to the shift of the yellow matrix, which is depicted on the right. Only the overlapping parts (in pink) contribute to the computation for each output matrix element.

Listing 1 A trivial implementation of cross-correlation.

```
for (int i = -(n - 1); i <= n - 1; i++)
    for (int j = -(n - 1); j <= n - 1; j++)
        for (int y = 0; y < n - abs(i); y++)
            for (int x = 0; x < n - abs(j); x++)
                result[j][i] += a[max(0, -i) + y][max(0, -j) + x] *
                                b[max(0,  i) + y][max(0,  j) + x];
```

For general $w \times h$ matrices, the cross-correlation has a time complexity of $\mathcal{O}(w^2 \cdot h^2)$. Alternatively, the input can be modified and passed to the Fast Fourier Transform (FFT) with a more pleasing time complexity of $\mathcal{O}(w \cdot h \cdot \log(w \cdot h))$. However, the hidden multiplicative factor, which materializes as an overhead in the implementations of FFT, favors the original, definition-based approach for small problem sizes.

To assess the optimality of the trivial implementation, let us compute the arithmetic intensity of a single overlap computation. For n overlapped element pairs, we need to perform n fused multiply-add operations. Considering the elements are represented as 4B single-precision floating points, the arithmetic intensity equals to $\frac{n}{2n*4} = \frac{1}{8}$. In Section 1.2.1, we detail that the ops-per-byte ratio of the modern NVIDIA V100 GPU is around 15 for single-precision floats. Since $\frac{1}{8} < 15$, the implementation of such an algorithm will be highly memory-bound.

2.3.2 Optimization techniques

With this in mind, we experimented with extreme data reuse techniques. There are many opportunities where the same data is read multiple times:

- When computing neighboring overlaps, most data from left and right matrices are read multiple times.

- In the $1 : N$ scenario, when cross-correlating one left matrix with many right matrices, the data from the left matrix is read multiple times when computing the same overlap between multiple right matrices.

- In the $N : M$ scenario, multiple left and right matrices can be read once to compute overlaps between all of them.

With such techniques, the arithmetic intensity is theoretically unbounded, provided an infinitely big input. However, the physical resources limit the practical values of the reuse factor. The most important resource is the register file. In order to reuse data, threads need to have them stored in the registers. The number of registers is limited, and the compiler can spill the data into slower memory if the limit is reached. Also, if the amount of resources per thread is high, the occupancy of the GPU can drop, and the achievable performance can decrease.

The other big issue is the parallelism. Naturally, if one worker performs multiple operations for the sake of sharing, the parallelism decreases. Therefore, the most effective implementation maintains the optimal balance between data sharing and parallelism.

Lastly, in Section 1.2.2 we described that the memory hierarchy can aid a low arithmetic intensity: the memory bandwidth is inversely proportional to the ops per byte ratio. For NVIDIA V100, the ops per byte ratio is $3.5\times$ less when using L2 cache and $13\times$ less when using shared memory, compared to the off-chip memory [49]. As a result, if the caches are utilized correctly, less data sharing is required to utilize the hardware capabilities fully.

In our implementation, we took GPU-specific advantage of even faster memory than shared memory, register file itself, to cache the data. The GPU register file is rather big, having 65536 32B registers. However, the caching possibilities on thread registers are very limited. They are only allowed on a warp level and allow only a specific access pattern. Still, they provide a higher throughput than the closest cache, the shared memory.

2.3.3 Results

The final implementation of the cross-correlation algorithm achieved a speedup of $5 - 10\times$ over a naïve GPU code. The benchmark also uncovered the boundaries, above which a more expensive time complexity of the definition-based algorithm overcomes the overhead of the FFT approach. In summary, achieved

speedup can enable physicists to analyze the data at a similar rate as EBSD cameras produce.

2.4 Stochastic simulation of Boolean networks

In Systems biology, the analysis of large and complex biological systems is a challenging task. *Boolean models* [50] have gained popularity due to their simplicity, scalability, and ability to describe complex signaling networks. A Boolean model consists of n nodes with binary values — active or inactive. A node represents an event in the system, such as a gene being expressed or a protein being activated. The complete state of the system is, therefore, defined by a Boolean word composed of n bits. The states can transition within each other based on n Boolean formulae, one for each node. Given an initial state, the task of the model is to predict the probability distribution of system outcomes after a specified amount of time.

Numerical methods, such as ExaStoLog [51], have been developed and used to solve Boolean models; however, they are limited to relatively small models (tens of nodes) due to the exponential memory requirements. For larger models, it has proved to be much more efficient to approximate the results by simulation. A C++ software, MaBoSS [52], simulates these systems by applying the kinetic Monte-Carlo algorithm [53] on the Boolean network. In summary, the MaBoSS tool simulates millions of random walks (also called the *trajectories*) on the state space of the Boolean network (a directed weighted graph). Algorithm 3 provides a high-level view of how a trajectory is sampled. The trajectories are then aggregated in a histogram-like fashion to compute various statistics, such as the probability distribution of the states over time, as is shown in Figure 14.

Algorithm 3 A single iteration of the MaBoSS simulation of a trajectory, given the trajectory state S at time t.

1: **procedure** TRAJECTORYSIMULATIONSTEP(S, t)
2: Compute transition probabilities $p_1, \dots, p_n$ by evaluating the Boolean formulae $\mathcal{B}_1, \dots, \mathcal{B}_n$ on S
3: Sample the next state S' and transition time Δt according to the probabilities $p_1, \dots, p_n$
4: **return** $S', t + \Delta t$
5: **end procedure**

Due to the stochastic nature, this approach can overcome the limitations of numerical methods and has enabled the analysis of bigger models, running instances with a few hundred nodes in a reasonable amount of time. On the other

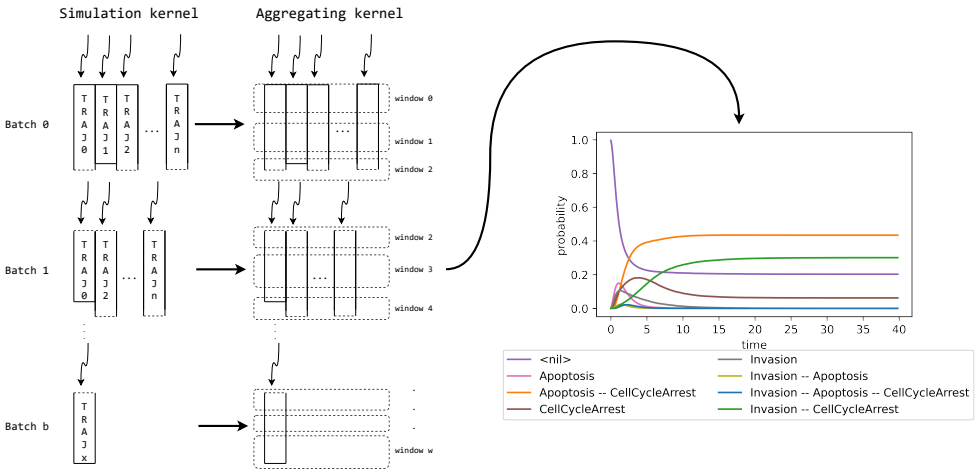


Figure 14 The MaBoSS projection of simulated trajectories averaged over a specific time window.

hand, allowing even bigger inputs would require billions of simulated trajectories in order to obtain a reliable result, which puts significant pressure on the tool and its ability to scale well. A parallel CPU MaBoSS simulation of a relatively small Sizek model [54] ($n = 93$) with 10^6 trajectories takes the order of minutes on a high-end computer. Simulating models with thousands of nodes and billions of trajectories would require an order of days in runtime, hindering further analysis of the models.

Another performance challenge is the so-called *mutant analysis*. In an existing model, a node is *mutated* to a static value, losing the ability to change its state. This technique is used to predict the effect of a gene knockout or overexpression (a node value immutably set to 0 or 1, respectively). During the analysis, the same model is run $2n$ times to gain the predictions for all *single* mutations (both suppressed and expressed). Naturally, this approach can be extended to research the effect of multiple mutants — running the model with multiple nodes mutated at once. However, just single mutants analysis has been considered computationally expensive, subjecting double or triple mutants to a supercomputer-level task.

2.4.1 GPU acceleration challenges

Although the parallelization of MaBoSS may seem straightforward, it includes challenges that can significantly affect the overall performance. The main issue is the traversal of the binary expression tree to evaluate the Boolean formulae

(Algorithm 3, Line 2). Generally, traversing irregular data structures causes conditional branching and creates an irregular data access pattern (refers to Thread divergence and Memory coalescing in Section 1.2). A combination of these issues may leave GPU heavily underutilized, gaining very little to no performance improvement over a well-parallelized CPU code.

Thread divergence can be mitigated by cleverly distributing formulae evaluations among the threads. The most straightforward approach is to assign all n formulae computations to a single thread such that the threads execute the same formula in each step. The divergence within the evaluation of the same formula can be solved by turning off short-circuiting optimization and enforcing the threads to evaluate the whole expression uniformly. The other possible approach would be to assign one formula to a thread. Naturally, this enables more parallelism but requires a more complex data structure to prevent thread divergence. The formulae need to be represented in the code so that each of their execution is issued as the same sequence of instructions. A way to achieve this is to convert formulas to CNF or DNF form and store them in the memory as a vector of bitmasks with a unified length.

In our work, we partly followed the first approach. But instead of designing an ideal memory layout for the formulae, we used the *CUDA NVRTC* [55] runtime compilation library to compile the formulas on the fly. It has the benefit of encoding the formulas into a native binary code, removing the necessity to fetch the data from the memory each time the formula is evaluated. Apart from that, the compiler can run a set of optimizations on the code, further improving the performance. Consequently, the runtime compilation comes at the cost of an additional runtime. Thankfully, our benchmarking showed that the compilation time amortizes well for simulations with reasonably big inputs.

2.4.2 Results

The final implementation of the MaBoSS simulation achieved a speedup of $100-300\times$ on real-world datasets over the parallel CPU version. Moreover, we created a synthetic test suite to stress the scalability of the implementation, which showed that the GPU implementation can simulate models with thousands of nodes and billions of trajectories in minutes. These results suggest that the consumer laptop equipped with a GPU can simulate models that were previously restricted to high-end computers. Furthermore, the double and triple mutant analysis no longer belongs just to the domain of supercomputers but could become feasible on data center-grade GPUs. Overall, we believe that the GPU acceleration of the MaBoSS simulation can significantly improve the analysis of Boolean networks and enable researchers to explore complex models while conveniently using their personal computers.

2.5 Summary

We have presented the work of implementing four scientific algorithms in the HPC domain. Overall, the results have shown that employing GPUs to solve scientific problems can help researchers push the boundaries of the size, scale, and complexity of analyzed data while leveraging the computational power of their GPU-equipped personal computers or laptops.

Further summarizing the optimization used, it comes as no surprise that the most important aspect to consider when designing an optimized implementation is memory. Memory complexity, memory access patterns, memory hierarchies, and data caching repeatedly occurred in our implementation designs and have contributed the most to performance improvements. This is yet supported by the fact that the gap between computational power and memory bandwidth tends to widen [56], causing more algorithms to be memory-bound, making memory optimizations even more crucial in the future.

Therefore, we dedicate the second chapter of this thesis to the methods for streamlining memory-related optimizations, which we collected during our optimization efforts and found them to be effective in aiding HPC programmers in writing optimized CPU or GPU code from scratch.

Chapter 3

Streamlining the development of parallel algorithms using Noarr

Commonly, the core computation of scientific algorithms is centered around a master data structure. The examples include a matrix composed of cell features in an agent-based simulator [57], a grid of substrates in a diffusion solver [58], a transition rates graph for Markov processes [51], etc. In the vast majority of these algorithms, the location of the most performance-critical parts lies in a nested loop over these data structures.

Possibly due to the lack of hardware expertise, usually there is no special attention paid to the memory performance when such loops are written by the domain experts [59]. As a result, the programs show poor data locality, spending most of the CPU cycles waiting for the memory pipeline to deliver the data.

A plethora of research works confirm that changing the way how data is laid in memory or modifying the nested loops can result in a significant performance improvement [60–62]. Such optimizations may include decreasing the number of cache misses by grouping the operation on the same data together, employing prefetching by streamlining the memory access pattern, or utilizing the vector instructions by aligning the data in memory.

Arguably, given a nested loop to optimize, the expert in the performance optimization domain can pinpoint the biggest bottlenecks w.r.t. memory and propose close to optimal modifications with great probability. The most common modifications would include reordering the loops for the most possible serial access pattern and dividing the loops into tiles or strides to employ cache hierarchies [63]. The problematic activity here, which takes the most programmers time, is putting these modifications into code. For example, let us have a nested loop of depth 3 with control variables `i, j, k` and bounds `I, J, K`. A simple tiling modification of the loop adds complexity in loop depth, variable count, and index computation:

```
1  for (i1 = 0; i1 < I / tile_I; i1++)
2    for (j1 = 0; j1 < J / tile_J; j1++)
3      for (k1 = 0; k1 < K / tile_K; k1++)
4        for (i2 = 0; i2 < tile_I; i2++)
5          for (j2 = 0; j2 < tile_J; i2++)
6            for (k2 = 0; k2 < tile_K; k2++)
7            // i=i1*tile_I+i2; j=j1*tile_J+j2; k=k1*tile_K+k2;
```

Omitting corner cases, the code is already verbose and rather error-prone (a careful eye may spot a mistake, a usual copy-paste bug on Line 5). Naturally, the complexity of the code grows with the complexity of the optimization.

Thankfully, many tools have been developed to alleviate this issue. Some provide automatic optimizations built within a compiler [64, 65], some extend a compiler with pragma-like annotations to guide the optimization process [66–69] and others include ad-hoc solutions for a specific family of algorithms [70, 71]. However, very little attention has been paid to aiding HPC programmers in writing the optimized code from scratch by providing them with a clean and maintainable way to express complex loop traversal and memory layout fit for their specific problem.

To alleviate and streamline the mundane programming tasks repeatedly encountered during our work on optimizing scientific algorithms, we focused our work on developing a C++ library *Noarr*. The main benefit of the library is that it allows to expressively and extensively define memory layouts of regular n-dimensional arrays and provides loop transformation primitives for their optimal traversal. It empowers HPC experts to swiftly develop their optimizations as a clean, maintainable code open to autotuning and parallelism while staying within the borders of the C++ standard without adding any dependency on the final software product (which is an advantage for the deployment in a scientific community).

3.1 Memory Layouts

Generally, the way how data is laid in memory, a *memory layout*, can be portrayed as a projection of its index space to a linear memory space. If we limit ourselves to a general (non-ragged) n-dimensional array, we can define memory layout mathematically as follows:

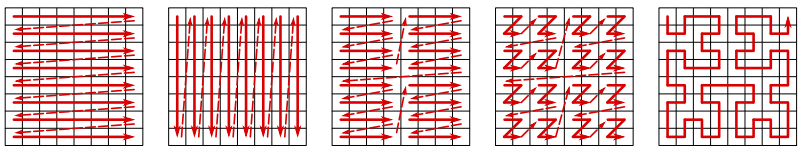
Definition 2 (Memory Layout)**.** *Suppose a regular n-dimensional array A with dimension lengths $d_1, \dots, d_n$ and an index space $\mathcal{I} = \{0, \dots, d_1 - 1\} \times \cdots \times \{0, \dots, d_n - 1\}$. We define a* memory layout $\mathbb{L}$ *of A as a bijection $\mathbb{L} : \mathcal{I} \to \{0, \dots, \prod_i d_i - 1\}$.*

Perhaps the most common memory layouts are row-major and column-major of a matrix (Figures 15a and 15b). Their respective bijections can be generated by indexing functions $L_{\text{row}}(i, j) = i \cdot d_2 + j$ and $L_{\text{col}}(i, j) = j \cdot d_1 + i$ and these functions would be carried to source code with slight modifications.

As alluded to in the motivation, common layouts used to optimize memory accesses require a complex code. A tiled matrix layout (Figure 15c), the layout paramount for some algorithms to optimally use cache hierarchies, already requires a verbose indexing function. Helping ourselves with adding two intra-tile dimensions, the indexing function would look as follows:

$$L_{\text{tiled}}(i, j, k, l) = (((i \cdot d_2 + j) \cdot d_3) + k) \cdot d_4 + l$$

Arguably, adding another tile dimension (to utilize multiple levels of cache), swapping intra-tile layout to column-major form, or implementing layouts as space-filling curves (Figures 15d and 15e, especially useful when dealing with cache-oblivious algorithms [72]) becomes less and less trivial. Moreover, considering the indexing functions are written ad-hoc, the layouts are tough subjects to change since a layout change requires a thoughtful and error-prone rewrite of all index function occurrences. On top of that, a complex indexing function is far from self-describing, making it hard, even close to impossible, to guess the layout intentions.



(a) row-major **(b)** col-major **(c)** tiled **(d)** z-curve **(e)** Hilbert curve

Figure 15 Instances of various matrix layouts.

3.1.1 Background

The importance of these issues has been recognized by the HPC community, especially by the authors of HPC programming frameworks. Perhaps the oldest example is Kokkos [73], a platform-agnostic programming model that defined a memory layout as a *first-class object*. The layout is defined as a vector of dynamic dimension lengths; the dimensions are laid out in memory from left to right (the leftmost dimension being laid in the memory continuously with a stride of 1, also called Fortran-style), from right to left (perhaps the most used one, C-style) or generally by a custom vector of strides. Such a simple approach covers a wide range of HPC use cases and has been adopted by other frameworks, such as

GridTools [71]. An example of defining a layout with 3 dimensions of lengths 3, 3 and 4 and their respective strides 1, 5 and 20 in Kokkos is as follows:

```
1  // Some storage
2  int* ptr = new int[80];
3  // A layout object with pairs of size and stride
4  Kokkos::LayoutStride layout(3, 1, 3, 5, 4, 20);
5  // A view from the pointer and the layout
6  Kokkos::View<int***> w(ptr, layout);
7  int elem = w(0, 0, 0);
```

Line 6 highlights another useful feature of the framework: The ability to decouple the layout from the underlying memory. The layout is a separate object from the memory pointer, which allows for easy reuse of the layout in different parts of the code. Connecting a layout to a memory is done explicitly by wrapping a layout and a memory pointer into a `View` object.

Recently, much bigger communities have started to invest time in designing an extensible way of defining layouts. C++ has standardized *mdspan* in C++23. It takes a finer approach, allowing to define layout dimensions using a more complex extent structure. Such a structure enables defining some dimension lengths as static (known at compile time) and mixing them with some dynamic (known at runtime). With such information during compile-time, a compiler can employ optimizations such as constant folding, loop unrolling, or automatized vectorization. Similarly, as with a vector of strides in Kokkos, it defines a `LayoutPolicy`, responsible for converting dimensions to underlying 1D memory, and `mdspan` class as a `View` alternative:

```
1  // Some storage
2  int* ptr = new int[80];
3  // An extent structure, the first two dims are static
4  std::extents<3, 3, std::dynamic_extent> ext(4);
5  // mdspan object with a pointer, extents and Fortran-style layout policy
6  std::mdspan<int, decltype(ext), std::layout_left> s(ptr, ext);
7  int elem = s[0, 0, 0];
```

Nvidia has recently introduced *CuTe*, a collection of C++ CUDA template abstractions for defining and operating on hierarchically multidimensional layouts of threads and data [74]. It follows similar principles as with the C++ standard: it defines *shape*, *stride*, and *tensor* as alternatives to Extents, LayoutPolicy, and mdspan. The same example as above written using CuTe lists as follows:

```
1  // Some storage
2  int* ptr = new int[80];
3  // A layout as a pair of shape and stride
4  Layout l = make_layout(
5    make_shape(Int<3>(), Int<3>(), 4),
6    make_stride(Int<1>(), Int<3>(), Int<9>())
7  );
8  // A tensor object with a pointer and a layout
```

```
Tensor t = make_tensor(ptr, l);
int elem = t(0, 0, 0);
```

Shapes and strides can be nested, creating a hierarchy of coordinates. Also, every layout (even a nested one) can be indexed with 1D coordinates, iterating the index space in a colexicografic order. This allows for a natural indexation of complicated layouts; e.g., a tiled matrix 4D layout can be indexed using standard 2D coordinates, hiding the complexity of the layout. Apart from that, CuTe also takes a formal approach, creating a *layout algebra*. It defines a set of operations over layouts, such as concatenation, composition, tiling, and others. These operations are used to build more complex layouts from simpler ones.

3.1.2 Noarr Layouts

Noarr aims to enhance the expressiveness, extendibility, and maintainability of the code. The main points which distinguish Noarr from other libraries are:

- *Named dimensions* — the dimensions are not defined just by an order of shape or stride vectors, but they are named. This allows to query a dimension regardless of its global index.
- *Proto-structures* — a set of building blocks is provided to allow to easily define complex layouts by their various composition.

Let us give an example of a row-major matrix memory layout using Noarr. Considering `'i'` as a row dimension and `'j'` as a column dimension, the layout can be defined as follows:

```
1 // Some storage
2 int* ptr = new int[80];
3 // A row-major matrix layout
4 auto row_l = scalar<int>() ^ vector<'j', 'i'>(lit<3>, 4);
5 // A bag object with a pointer and a layout
6 bag b = make_bag(row_l, ptr);
7 int elem = b.at<'i', 'j'>(0, 0);
```

Although Noarr predates mdspan and CuTe, it shares many similarities: the separation of layout and underlying memory (using `bag` on Line 6) and the ability to define static and dynamic-sized dimensions (using `lit` on Line 4). Perhaps the biggest difference is the absence of a stride vector. In Noarr, the strides are computed automatically according to the order of the named dimensions. The matrix layout above can be changed to column-major by a simple named dimensions swap:

```
auto col_l = scalar<int>() ^ vector<'i', 'j'>(4, lit<3>);
```

The order in which the dimensions are defined signals the way how they are laid in memory, which is in the left-to-right fashion. The meaning of the dimension names is important here, as it determines the *logical order*. The order in the definition then defines the *physical order*. For example, let us have a cube layout, denoting its height, width and depth as `'i'`, `'j'` and `'k'` respectively. A C-style cube layout could be defined using `vector<'k', 'j', 'i'>()` protostructure. A custom layout (neither Fortran- nor C-style) can be simply written as `vector<'j', 'k', 'i'>()`. Using this logical–physical distinction, all of the vastly used stride vectors can be simulated by permuting the dimensions order.

The other aspect of Noarr is the way how the layout is defined — as a composition of building blocks, the protostructures. We already mentioned `row_l`, which is a composition of `scalar<int>` and `vector<'j', 'i'>`, which, when composed, represents a vector (`'i'`) of vectors (`'j'`) of integer scalars (`int`) — an integer matrix. The left-to-right reading of the layout definition allows for easy extension of the existing layout. Let us enumerate examples of more complex layouts in the following listing.

```
// A layout for a batch of row-major matrices
// and offset to the 1st row of the 2nd matrix
auto batched_l = row_l ^ vector<'b'>(10);
size_t o1 = batched_l | offset<'i', 'j', 'b'>(1, 0, 2);
// Two tiled matrix layouts (both row-major and col-major,
// where 'k' and 'l' denote tile rows and columns)
auto tiled_rr_l = row_l ^ vector<'l', 'k'>(5, 10);
auto tiled_rc_l = row_l ^ vector<'k', 'l'>(10, 5);
// A sublayout composed of the 3rd row tiles
auto tiled_3rd_l = tiled_rc_l ^ fix<'k'>(3);
// A tiled matrix layout indexable using 2 coordinates
auto tiled_2d_l = tiled_rr_l ^ merge_blocks<'i', 'k', 'I'>() ^
↪  merge_blocks<'j', 'l', 'J'>();
```

The protostructures allow for expressing complex layouts in a readable and verifiable way. Furthermore, their extensibility in composition and separation from the underlying memory also allows for plug-in layouts, which can be easily *reused* in different parts of the code:

Let us have a complex layout of a multi-layered 3D grid gradient, which occurs on various code places and provides layouts for multiple memory pointers. In that case, a layout can be declared once with sufficient scope and reused in places where needed:

```
// A globally accessible layout
auto gradient_layout = scalar<float>()
  ^ vector<'s', 'x', 'y', 'z', 'g'>(
    substrates, grid_dims[0], grid_dims[1], grid_dims[2], lit<3>
  );

```

```
7   void func1_cpu(float* grad_ptr) {
8     bag b = make_bag(gradient_layout, grad_ptr);
9     // use the layout on cpu memory
10  }
11
12  void func2_gpu(float* global_mem_ptr) {
13    bag glb_b = make_bag(gradient_layout, global_mem_ptr);
14    bag shm_b = make_bag(gradient_layout, shared_mem_ptr);
15    // use the layout on gpu global and shared memory
16  }
```

The other benefit of having a layout as a reusable first-class object is the ability to have a localized change when deciding to modify the layout code during the implementation. Extending our previous example, if we decide to reorder the gradient dimension `'g'` to be the innermost one, we only change the definition of the object on Line 2 — all the data structures using the layout will be transparently updated without any further code changes.

3.2 Traversing Layouts

Having specified how a data structure is laid in memory, the other possible point of optimization lies in the order how the data structure is iterated, its *traversal*. In general, a traversal is usually written as a sequence of nested loops, which in turn generates a sequence of data structure accesses. Using such interpretation, we define the traversal of a layout as follows:

Definition 3 (Layout Traversal)**.** *Suppose a sequence of n nested loops with the loop boundaries $d_1, \dots, d_n$ and a memory layout of a data structure with an index space $\mathcal{I}$. We define a* layout traversal $\mathbb{T}$ *as a bijection* $\mathbb{T} : \{0, \dots, \prod_i d_i - 1\} \to \mathcal{I}$.

In conjunction with the memory layout from Definition 2, a natural observation arises: $\mathbb{L}$ and $\mathbb{T}$ are composable, and their composition $\mathbb{L} \circ \mathbb{T}$ is a function *from integers to integers* — it generates the memory access pattern. As a corollary, the same sequence of accesses can be generated by either changing the layout ($\mathbb{L}$) or its traversal ($\mathbb{T}$). Therefore, as with a layout, modifying a traversal can improve data locality, both in time and space. Although modifying traversal can be constrained by data dependencies, loop transformation is still a well-researched topic and complements the memory layout optimization [59].

Writing complex traversals complements the issues discussed above when detailing the layouts. So, a tool that simplifies this process can share the same benefits: expressivity, maintainability, and reusability. Moreover, since traversing over some master data structures is the common place where parallelism is introduced in the code, the abstraction of loop transformation can further extend to parallel processing.

3.2.1 Background

Quantitatively, *loop transformation* is perhaps a more researched topic than layout transformation. Neither library mentioned above provides ways to transform a traversal, but many other tools do. Contemporary compilers employ sophisticated loop optimizers based on the polyhedral model, such as Graphite in GCC [64] or Polly in LLVM [65]. However, these optimizers are limited by the lack of information specified by the user in the code.

For more user-guided approaches, there are annotation-based tools, such as Poet [67], Chill [68] or Loopy [69]. These tools allow to specify the traversal transformations by adding special comments or pragmas to the code. The transformations are then applied by a precompiler, which generates the optimized code. Perhaps the best representative of annotation-based frameworks is done by Kruse et al. [75]. The authors have presented pragma-based user-directed loop transformations for the Clang compiler, which has been partly adopted by the OpenMP standard. The transformations include loop tiling:

```
// original code
#pragma clang loop(i,j) tile sizes(4,8)
for (int i = 0; i < n; i+=1)
  for (int j = 0; j < m; j+=1)
// transformed code
for (int i1 = 0; i1 < n; i1+=4)
  for (int j1 = 0; j1 < m; j1+=8)
    for (int i2 = i1; i2 < n && i2 < i1+4; i2+=1)
      for (int j2 = j2; j2 < m && j2 < j1+8; j2+=1)
```

or loop reordering:

```
// original code
#pragma clang loop(i,j) interchange permutation(j,i)
for (int i = 0; i < n; i+=1)
  for (int j = 0; j < m; j+=1)
// transformed code
for (int j = 0; j < m; j+=1)
  for (int i = 0; i < n; i+=1)
```

but also loop fusion, fission, reversal, and others [76].

These tools are straightforward to use, and their ability to combine them to form complex loop transformations makes them very expressive. A minor downside is that they are typically closed to extensions, and for most of them, a precompiler with an extra preprocessing step or a custom compiler extension is needed.

On the other hand, the target problems for annotation-based loop transformation tools are somehow limited. Naturally, the best use of them is made when there is already a baseline implementation, and the optimization is achieved just by adding a few lines on the top of the loop with the hottest performance spots. However, when developing an optimized solution from scratch and when the

scale of the program passes a certain complexity threshold, optimizing just by annotations may become cumbersome.

A good evidence of this fact is case studies that compared the effort of programming a parallel algorithm using OpenMP, also a pragma-based library, with pure C++ libraries, such as TBB. The studies generally confirm that ease of expressing parallelism in OpenMP is traded for a lower abstraction level and flexibility. Language-based approaches, such as TBB, can utilize object-oriented design for more control, fostering of good programming style and higher abstraction level for *newly* developed parallel programs [77–79].

3.2.2 Noarr Traversers

The similarities in memory layout and its traversal led to the development of *Noarr Traversers*. It is based on the idea that every single loop, in some sequence of nested loops, can be interpreted as one (named) dimension in an index space domain. Assuming perfectly nested loops, such traversal can be interpreted as a Noarr layout and be subjected to the same transformations as a memory layout. To sum up, the key points of Noarr Traversers are as follows:

- *Named dimensions* — a traverser comprises named dimensions, each representing a loop in a sequence of nested loops.
- *Proto-structures* — exactly as with layouts, a traverser is subject to an extension using the same set of proto-structures.
- *Index space extraction* — the traverser extracts dimensions from the passed-in layouts and yields an index space corresponding to the cartesian product of the extracted dimensions.

Let us highlight the main idea of Noarr Traverser on an example of matrix multiplication, where a plain C++ implementation would look like this:

```
for (int i = 0; i < m; i++)
  for (int j = 0; j < n; j++)
    for (int k = 0; k < p; k++)
      C[i][j] += A[i][k] * B[k][j];
```

The nested loops follow the order of 3 present dimensions: i, j, and k denoting rows of C and A, columns of C and B, and rows of A and columns of B, respectively. Smartly selecting the named dimensions of the layouts, the traversal can be rewritten using Noarr as follows:

```
auto A = make_bag(ptr_a, scalar<int> ^ vector<'k', 'i'>(K, I));
auto B = make_bag(ptr_b, scalar<int> ^ vector<'j', 'k'>(J, K));
auto C = make_bag(ptr_c, scalar<int> ^ vector<'j', 'i'>(J, I));

```

```
5   traverser(A, B, C) | [=](auto state) {
6     C[state] += A[state] * B[state];
7   };
```

The traverser accepts the layouts (or bags) it shall iterate over. It extracts 3 unique dimensions `'i'`, `'j'` and `'k'` and iterates over the index space composed of these dimensions as if they were written as a sequence of nested loops (which is equivalent to the cartesian product of these dimensions). The lambda function on Line 6 is then executed for each point in the index space, passing the tuple of indices as the `state` argument to the lambda.

Due to the same concept of named dimensions, traverser and layout objects are *isomorphic*. Consequently, changing the traversal order is done the same way as modifying the layout — by applying the proto-structures to the traverser. E.g., to reorder the loops to a more cache-friendly `'i'`, `'k'`, `'j'` order, the traversal can be rewritten as follows:

```
1   traverser(A, B, C) ^ reorder<'i', 'k', 'j'>() | [=](auto state) {
2     C[state] += A[state] * B[state];
3   };
```

Furthermore, a side product of the isomorphism is that a proto-structure can be applied either to the layout or to the traverser, granting the same effect.

Let us conclude the benefits of tracking loops by names with a more complex example: Some transformations change the nesting level of the underlying loops, e.g., during a strip-mine transformation, where a loop is split into two nested loops by a blocking factor [76]. Following the Noarr paradigm of named dimensions, a strip-mine is well defined by a proto-structure `into_blocks`, which accepts the first dimension as the input, which should be present in the layout, and two output dimension, which will be newly added to the composed layout. Since all loops are tracked by their names, the user has full control over the output of a transformation and can safely chain multiple transformations together. Tiled matrix multiplication can be expressed as follows:

```
1   // breaking dimension 'i' into two new dimensions 'I' and 'x'
2   auto tiles = into_blocks<'i', 'I', 'x'>(noarr::lit<16>)
3              ^ into_blocks<'j', 'J', 'y'>(noarr::lit<16>)
4              ^ into_blocks<'k', 'K', 'z'>(noarr::lit<16>)
5              // using the new dimensions
6              ^ reorder<'I', 'J', 'K', 'x', 'y', 'z'>();
7
8   traverser(A, B, C) ^ tiles | [=](auto state) {
9     C[state] += A[state] * B[state];
10  };
```

3.2.3 Reusability

Similarly as with layouts, a specific traversal order can be defined as an object, such as `tiles` on Line 2 in the previous code sample, and provide the same benefits: locality of change and reusal.

But this feature further extends to a perhaps more powerful concept, which we call *layout agnosticism* and *traversal agnosticism*. As a case of the separation of concerns, a memory layout and traversal can be extracted from a function to the level of function arguments. These arguments may then serve as interfaces for the concrete traversal or layout objects:

```
1 template <class A_t, class B_t, class C_t, class Order_t>
2 void matmul(A_t& A, B_t& B, C_t& C, Order_t my_order)
3 {
4     traverser(A, B, C) ^ my_order | [=](auto state) {
5         C[state] += A[state] * B[state];
6     };
7 }
```

The function `matmul` becomes agnostic to the layout or traversal specified, and, regardless of the passed-in objects, it will run the same operations, just in a different order.

The modular template-based design of Noarr allows us to mimic the auto-tuning systems, which search the vast space of transformation, trying to find the most performant one for a specific computational function. In our approach, we may be able to express precisely that with the traversal-agnostic and layout-agnostic functions.

3.2.4 Parallelism

Since Noarr is general enough to allow the division of traversals into independent sub-traversals, the user can also guide the parallelism of a program. Proto-structures such as `fix`, `slice` or `step` provide expressive ways to split a work over a data structure into custom sections. The user can then assign each section to a separate thread, which will iterate over the section independently.

But if the user wants to offload this task to a well-known field-tested parallel library, Noarr is also open to this extension. Our modular design allows us to plug in various *parallel executors*. So far, we experimented with OpenMP, TBB, and CUDA, extending the library with parallel-for, parallel-reduce, and others. The parallel `for-each` using Noarr and OpenMP can be written as follows:

```
1 // parallelising over dimension x
2 #pragma omp parallel for
3 for (auto t_inner : a_traverser ^ hoist<'x'>()) {
4   t_inner | [=](auto state) { /* ... */}; // some independent work
5 }
```

The code sample further strengthens the benefit of named dimensions: The user can directly specify the dimension to parallelize over. This is even more visible when targeting parallel accelerators (such as GPUs) in which a programmer can select multiple dimensions to utilize thread hierarchies (as described in Section 1.1):

```
auto A = make_bag(ptr_a, scalar<int>() ^ vector<'x', 'y'>(32, 1000));
// spawns a cuda block for each element of the 'y' dimension
// with as many threads as the size of 'x' dimension
auto ct = noarr::cuda_threads<'y', 'x'>(traverser(A));
a_kernel<<<ct.grid_dim(), ct.block_dim()>>>(ct.inner(), A);

__global__ void a_kernel(auto traverser, auto A) {
  // A is accessed according to the block index
  // and the thread index of the currently executing thread
  auto var = A[traverser.state()];
}
```

3.3 Summary

With the arsenal of proto-structures, Noarr provides options to specify many complex layouts and traversals. The object-oriented design of the library allows writing complex memory optimizations in an extensible, reusable, and even layout-agnostic way, with very little space for errors. The library has already been incorporated into our ongoing work of implementing a high-performance version of BioFVM [58], a diffusion solver for 3D biological simulations, aiding in various loop transformations, vectorization, dividing the data into subdomains for parallelization, and other optimizations, all realized in a clean and self-documenting way[1].

We believe that our solution will add a new expressive tool to a high-performance programming toolset, simplifying the research focusing on tuning of prewritten computational kernels and building complex optimized parallel algorithms from scratch.

[1] The code can be reviewed in the GitHub repository https://github.com/asmelko/paraBioFVM

Conclusion

This thesis summarizes several efforts to improve the performance of complex algorithms with inefficient implementations from contemporary scientific domains, improving their practical usability by applying high-performance computing principles. In the two chapters, it outlines the motivations and results of six research contributions, presented in Appendix A.

The first four contributions detail the parallelization and optimization challenges of the scientific algorithms which we have worked on during our research. The results include GPU applications that use novel data structures, promote high scalability, and utilize complicated GPU optimization techniques. The presented implementations provide orders of magnitude speedups over the prior state-of-the-art, enabling domain scientists to finish their analyses in minutes instead of days, process more data with greater accuracy, interactively visualize the results, and explore previously unattainable problem variations. The methodology of the works can serve as a helpful support for other researchers in the domain of GPGPU computing.

The remaining contributions present the use cases of the novel HPC library Noarr, specializing in expressing the layout and traversal of n-dimensional arrays, which are the most commonly used data structures in scientific computing. The novel approach of assigning names to the dimensions of the arrays and expressing the layout and traversal in a declarative way allows users to deploy memory-related optimizations in a more readable and maintainable manner while the library takes care of the complex indexing and loop transformations.

As mentioned throughout the thesis, the use of GPUs for general computing is becoming increasingly popular. Consequently, these devices improve in versatility with each addition of new core types, specialized high-throughput instructions, and new thread hierarchies. This creates unique opportunities for interesting future research: The implementation of Mahalanobis Hierarchical Clustering can be directly extended with the use of tensor cores, which may provide an additional order of magnitude performance improvement thanks to their high compute bandwidth. Further, we plan to continue on k-NN development, which we have already started during the work on EmbedSOM; we believe that

increasing the caching capabilities by employing the new feature of distributed shared memory may improve the data throughput of such a highly memory-bound algorithm. Regarding the following research on Noarr library, we would like to continue extending the tool in the auto-tuning direction, ultimately providing a machine learning-guided tuning of the array layouts and traversals. We also plan to finish our work on biological simulations, especially BioFVM and PhysiCell, which has already served as a valuable case study for Noarr and has partly guided its development thanks to the presence of multiple complex high-dimensional arrays, which accelerated its usability.

Finally we hope that the contributions of this thesis will help to address the challenges posed by the increasing complexity of GPUs and the widening gap between compute and memory bandwidth. We believe that the presented methodologies and contributions provide valuable insights and can serve as a foundation for further research in the field of GPGPU computing and HPC.

Bibliography

[1] Mahmoud Khairy, Amr G Wassal, and Mohamed Zahran. "A survey of architectural approaches for improving GPGPU performance, programmability and heterogeneity". *Journal of Parallel and Distributed Computing* **127** (2019), pp. 65–88.

[2] *TOP500*. 2024. URL: https://www.top500.org/ (visited on 06/06/2024).

[3] Sparsh Mittal and Shraiysh Vaishay. "A survey of techniques for optimizing deep learning on GPUs". *Journal of Systems Architecture* **99** (2019), p. 101635.

[4] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. "Gpu-accelerated database systems: Survey and open challenges". *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV: Selected Papers from ADBIS 2013 Satellite Events* (2014), pp. 1–35.

[5] T Kalaiselvi, P Sriramakrishnan, and K Somasundaram. "Survey of using GPU CUDA programming model in medical image analysis". *Informatics in Medicine Unlocked* **9** (2017), pp. 133–144.

[6] Jin Wang and Sudhakar Yalamanchili. "Characterization and analysis of dynamic parallelism in unstructured GPU applications". In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2014, pp. 51–60.

[7] Carl Pearson et al. "Evaluating characteristics of CUDA communication primitives on high-bandwidth interconnects". In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. 2019, pp. 209–218.

[8] Ruyi Qian, Mengjuan Gao, Qinwen Shi, and Yuanchao Xu. "An Empirical Study of Memory Pool Based Allocation and Reuse in CUDA Graph". In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2023, pp. 394–406.

[9] Dian-Lun Lin and Tsung-Wei Huang. "Efficient GPU computation using task graph parallelism". In: *Euro-Par 2021: Parallel Processing: 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1–3, 2021, Proceedings 27*. Springer. 2021, pp. 435–450.

[10] Jack Choquette. "Nvidia hopper h100 gpu: Scaling performance". *IEEE Micro* (2023).

[11] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. "Accelerating reduction and scan using tensor core units". In: *Proceedings of the ACM International Conference on Supercomputing*. 2019, pp. 46–57.

[12] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben Van Werkhoven, and Henri E Bal. "Optimization techniques for GPU programming". *ACM Computing Surveys* **55**.11 (2023), pp. 1–81.

[13] Guillem Pratx and Lei Xing. "GPU computing in medical physics: A review". *Medical physics* **38**.5 (2011), pp. 2685–2697.

[14] Cristobal A Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. "A survey on parallel computing and its applications in data-parallel problems using GPU architectures". *Communications in Computational Physics* **15**.2 (2014), pp. 285–329.

[15] John F Croix and Sunil P Khatri. "Introduction to GPU programming for EDA". In: *Proceedings of the 2009 International Conference on Computer-Aided Design*. 2009, pp. 276–280.

[16] Nvidia. *CUDA C++ Programming Guide*. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. 2024.

[17] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures". *Communications of the ACM* **52**.4 (2009), pp. 65–76.

[18] Wikipedia. *Roofline model — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=Roofline%20model&oldid=1193478914`. [Online; accessed 16-June-2024]. 2024.

[19] Nvidia. *How to Overlap Data Transfers in CUDA C/C++*. `https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/`. 2024.

[20] Nvidia. *Inside Volta: The World's Most Advanced Data Center GPU*. `https://developer.nvidia.com/blog/inside-volta/`. 2024.

[21] Jonathan R Brestoff and John L Frater. "Contemporary challenges in clinical flow cytometry: small samples, big data, little time". *The journal of applied laboratory medicine* **7**.4 (2022), pp. 931–944.

[22] Junwei Zhou, Qian Wei, Chao Wu, and Guangzhong Sun. "A High Performance Computing Method for Noise Cross-Correlation Functions of Seismic Data". In: *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE. 2021, pp. 1179–1182.

[23] Karel Fišer et al. "Detection and monitoring of normal and leukemic cell populations with hierarchical clustering of flow cytometry data". *Cytometry Part A* **81**.1 (2012), pp. 25–34.

[24] Miroslav Kratochvíl, David Bednárek, Tomáš Sieger, Karel Fišer, and Jiří Vondrášek. “ShinySOM: graphical SOM-based analysis of single-cell cytometry data”. *Bioinformatics* **36**.10 (2020), pp. 3288–3289.

[25] Prasanta Chandra Mahalanobis. “On the generalized distance in statistics”. In: National Institute of Science of India. 1936.

[26] P Dagnelie and A Merckx. “Using generalized distances in classification of groups”. *Biometrical journal* **33**.6 (1991), pp. 683–695.

[27] William HE Day and Herbert Edelsbrunner. “Efficient algorithms for agglomerative hierarchical clustering methods”. *Journal of classification* **1**.1 (1984), pp. 7–24.

[28] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. “Towards high performance paged memory for GPUs”. In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2016, pp. 345–357.

[29] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. “Batch-aware unified memory management in GPUs for irregular workloads”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 1357–1370.

[30] Raphael Landaverde, Tiansheng Zhang, Ayse K Coskun, and Martin Herbordt. “An investigation of unified memory access performance in cuda”. In: *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2014, pp. 1–6.

[31] *Flow Repository*. May 3, 2020. URL: https://flowrepository.org/id/FR-FCM-ZZPH.

[32] Laurens Van der Maaten and Geoffrey Hinton. “Visualizing data using t-SNE”. *Journal of machine learning research* **9**.11 (2008).

[33] Etienne Becht et al. “Dimensionality reduction for visualizing single-cell data using UMAP”. *Nature biotechnology* **37**.1 (2019), pp. 38–44.

[34] Ehsan Amid and Manfred K Warmuth. “TriMap: Large-scale dimensionality reduction using triplets”. *arXiv preprint arXiv:1910.00204* (2019).

[35] Miroslav Kratochvíl, Abhishek Koladiya, and Jiří Vondrášek. “Generalized EmbedSOM on quadtree-structured self-organizing maps”. *F1000Research* **8**.2120 (2019). [version 2; peer review: 2 approved], p. 2120.

[36] Nicola Pezzotti, Thomas Höllt, B Lelieveldt, Elmar Eisemann, and Anna Vilanova. “Hierarchical stochastic neighbor embedding”. In: *Computer Graphics Forum*. **35**. 3. Wiley Online Library. 2016, pp. 21–30.

[37] Nicola Pezzotti, Boudewijn PF Lelieveldt, Laurens Van Der Maaten, Thomas Höllt, Elmar Eisemann, and Anna Vilanova. “Approximated and user steerable tSNE for progressive visual analytics”. *IEEE transactions on visualization and computer graphics* **23**.7 (2016), pp. 1739–1752.

[38] George C Linderman, Manas Rachh, Jeremy G Hoskins, Stefan Steinerberger, and Yuval Kluger. “Efficient algorithms for t-distributed stochastic neighborhood embedding”. *arXiv preprint arXiv:1712.09005* (2017).

[39] Anna C Belkina, Christopher O Ciccolella, Rina Anno, Richard Halpert, Josef Spidlen, and Jennifer E Snyder-Cappione. “Automated optimal parameters for T-distributed stochastic neighbor embedding improve visualization and allow analysis of large datasets”. *BioRxiv* (2018), p. 451690.

[40] M Kratochvıl et al. “SOM-based embedding improves efficiency of high-dimensional cytometry data analysis”. *biorxiv* (2019).

[41] Mohamed Wahib and Naoya Maruyama. “Scalable kernel fusion for memory-bound GPU applications”. In: *SC’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2014, pp. 191–202.

[42] Daren Lee, Ivo Dinov, Bin Dong, Boris Gutman, Igor Yanovsky, and Arthur W Toga. “CUDA optimization strategies for compute-and memory-bound neuroimaging algorithms”. *Computer methods and programs in biomedicine* **106**.3 (2012), pp. 175–187.

[43] Pieter Ghysels, P Kłosiewicz, and Wim Vanroose. “Improving the arithmetic intensity of multigrid with the help of polynomial smoothers”. *Numerical Linear Algebra with Applications* **19**.2 (2012), pp. 253–267.

[44] Gangzhao Lu, Weizhe Zhang, and Zheng Wang. “Optimizing GPU memory transactions for convolution operations”. In: *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2020, pp. 399–403.

[45] Soňa Molnárová. “Throughput optimization of a multistage data visualisation pipeline” (2023).

[46] Rui Fan and Naim Dahnoun. “Real-time stereo vision-based lane detection system”. *Measurement Science and Technology* **29**.7 (2018), p. 074005.

[47] Jose A Belloch, Alberto Gonzalez, Antonio M Vidal, and Maximo Cobos. “On the performance of multi-GPU-based expert systems for acoustic localization involving massive microphone arrays”. *Expert Systems with Applications* **42**.13 (2015), pp. 5607–5620.

[48] Michal Bali. “Zpracování dat z elektronového mikroskopu pomocí GPU” (2021).

[49] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. “Dissecting the NVIDIA volta GPU architecture via microbenchmarking”. *arXiv preprint arXiv:1804.06826* (2018).

[50] Rui-Sheng Wang, Assieh Saadatpour, and Reka Albert. “Boolean modeling in systems biology: an overview of methodology and applications”. *Physical biology* **9**.5 (2012), p. 055001.

[51] Mihály Koltai, Vincent Noel, Andrei Zinovyev, Laurence Calzone, and Emmanuel Barillot. “Exact solving and sensitivity analysis of stochastic continuous time Boolean models”. *BMC bioinformatics* **21** (2020), pp. 1–22.

[52] Gautier Stoll et al. “MaBoSS 2.0: an environment for stochastic Boolean modeling”. *Bioinformatics* **33**.14 (2017), pp. 2226–2228.

[53] Gautier Stoll, Eric Viara, Emmanuel Barillot, and Laurence Calzone. “Continuous time Boolean modeling for biological signaling: application of Gillespie algorithm”. *BMC systems biology* **6** (2012), pp. 1–18.

[54] Herbert Sizek, Andrew Hamel, Dávid Deritei, Sarah Campbell, and Erzsébet Ravasz Regan. “Boolean model of growth signaling, cell cycle and apoptosis predicts the molecular mechanism of aberrant cell cycle progression driven by hyperactive PI3K”. *PLoS computational biology* **15**.3 (2019), e1006402.

[55] NVIDIA. *CUDA NVRTC*. 2023. URL: `https://docs.nvidia.com/cuda/nvrtc/index.html` (visited on 02/14/2024).

[56] Lingqi Zhang et al. “PERKS: a Locality-Optimized Execution Model for Iterative Memory-bound GPU Applications”. In: *Proceedings of the 37th International Conference on Supercomputing*. 2023, pp. 167–179.

[57] Ahmadreza Ghaffarizadeh, Randy Heiland, Samuel H Friedman, Shannon M Mumenthaler, and Paul Macklin. “PhysiCell: An open source physics-based cell simulator for 3-D multicellular systems”. *PLoS computational biology* **14**.2 (2018), e1005991.

[58] Ahmadreza Ghaffarizadeh, Samuel H Friedman, and Paul Macklin. “BioFVM: an efficient, parallelized diffusive transport solver for 3-D biological simulations”. *Bioinformatics* **32**.8 (2016), pp. 1256–1258.

[59] Philippe Clauss and Benoît Meister. “Automatic memory layout transformations to optimize spatial locality in parameterized loop nests”. *ACM SIGARCH computer architecture news* **28**.1 (2000), pp. 11–19.

[60] Zhangxiaowen Gong et al. “An empirical study of the effect of source-level loop transformations on compiler stability”. *Proceedings of the ACM on Programming Languages* **2**.OOPSLA (2018), pp. 1–29.

[61] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. “Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model”. In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. 2015, pp. 207–216.

[62] Matheus S Serpa et al. “Memory performance and bottlenecks in multicore and gpu architectures”. In: *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE. 2019, pp. 233–236.

[63] Michael E Wolf and Monica S Lam. “A data locality optimizing algorithm”. In: *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 1991, pp. 30–44.

[64] Konrad Trifunovic et al. “Graphite two years after: First lessons learned from real-world polyhedral compilation”. In: *GCC Research Opportunities Workshop (GROW'10)*. 2010.

[65] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. “Polly: performing polyhedral optimizations on a low-level intermediate representation”. *Parallel Processing Letters* **22**.04 (2012), p. 1250010.

[66] Sebastien Donadio et al. “A language for the compact representation of multiple program versions”. In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2005, pp. 136–151.

[67] Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan. “POET: Parameterized optimizations for empirical tuning”. In: *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2007, pp. 1–8.

[68] Chun Chen, Jacqueline Chame, and Mary Hall. *CHiLL: A framework for composing high-level loop transformations*. Tech. rep. Citeseer, 2008.

[69] Kedar S Namjoshi and Nimit Singhania. “Loopy: Programmable and formally verified loop transformations”. In: *International Static Analysis Symposium*. Springer. 2016, pp. 383–402.

[70] Christian R. Trott et al. “Kokkos 3: Programming Model Extensions for the Exascale Era”. *IEEE Transactions on Parallel and Distributed Systems* **33**.4 (2022), pp. 805–817. DOI: 10.1109/TPDS.2021.3097283.

[71] Anton Afanasyev et al. “GridTools: A framework for portable weather and climate applications”. *SoftwareX* **15** (2021), p. 100707. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2021.100707. URL: https://www.sciencedirect.com/science/article/pii/S2352711021000522.

[72] Michael Bader and Christoph Zenger. “Cache oblivious matrix multiplication using an element ordering based on a Peano curve”. *Linear Algebra and Its Applications* **417**.2-3 (2006), pp. 301–313.

[73] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns”. *Journal of Parallel and Distributed Computing* **74**.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202–3216. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003. URL: https://www.sciencedirect.com/science/article/pii/S0743731514001257.

[74] NVIDIA. *CUDA NVRTC*. 2024. URL: https://github.com/NVIDIA/cutlass/blob/main/media/docs/cute/00_quickstart.md (visited on 05/08/2024).

[75] Michael Kruse and Hal Finkel. “User-directed loop-transformations in Clang”. In: *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE. 2018, pp. 49–58.

[76] Kathryn S McKinley, Steve Carr, and Chau-Wen Tseng. “Improving data locality with loop transformations”. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **18**.4 (1996), pp. 424–453.

[77] Philipp Kegel, Maraike Schellmann, and Sergei Gorlatch. “Using openmp vs. threading building blocks for medical imaging on multi-cores”. In: *Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings 15*. Springer. 2009, pp. 654–665.

[78] Ensar Ajkunic, Hana Fatkic, Emina Omerovic, Kristina Talic, and Novica Nosovic. “A comparison of five parallel programming models for c++”. In: *2012 Proceedings of the 35th International Convention MIPRO*. IEEE. 2012, pp. 1780–1784.

[79] Peder Rindal Refsnes. “Comparison of Openmp and threading building blocks for expressing parallelism on shared-memory systems”. MA thesis. The University of Bergen, 2011.

Appendix A

Contributions

Contribution 1
Mahalanobis Hierarchical Clustering

Published as Adam Šmelko et al. "GPU-Accelerated Mahalanobis-Average Hierarchical Clustering Analysis". In: *European Conference on Parallel Processing*. Springer. 2021, pp. 580–595

GPU-Accelerated Mahalanobis-Average Hierarchical Clustering Analysis

Adam Šmelko[1(✉)], Miroslav Kratochvíl[1,2], Martin Kruliš[1], and Tomáš Sieger[3]

[1] Department of Software Engineering, Charles University, Prague, Czech Republic
smelko@ksi.ms.mff.cuni.cz
[2] Luxembourg Centre for Systems Biomedicine, University of Luxembourg, Esch-sur-Alzette, Luxembourg
[3] Department of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague, Prague, Czech Republic

Abstract. Hierarchical clustering is a common tool for simplification, exploration, and analysis of datasets in many areas of research. For data originating in flow cytometry, a specific variant of agglomerative clustering based Mahalanobis-average linkage has been shown to produce results better than the common linkages. However, the high complexity of computing the distance limits the applicability of the algorithm to datasets obtained from current equipment. We propose an optimized, GPU-accelerated open-source implementation of the Mahalanobis-average hierarchical clustering that improves the algorithm performance by over two orders of magnitude, thus allowing it to scale to the large datasets. We provide a detailed analysis of the optimizations and collected experimental results that are also portable to other hierarchical clustering algorithms; and demonstrate the use on realistic high-dimensional datasets.

Keywords: Clustering · High-dimensional · Mahalanobis distance · Parallel · GPU · CUDA

1 Introduction

Clustering algorithms are used as common components of many computation pipelines in data analysis and knowledge mining, enabling simplification and classification of huge numbers of observations into separate groups of similar data. Atop of that, a hierarchical clustering analysis (HCA) captures individual relations between clusters of data in a tree-like structure of dataset subsets (a *dendrogram*), where each subtree layer corresponds to a finer level of detail. The tree structure is suitable for many scenarios where the definition of clusters is unclear, such as in interactive analysis of noisy data where the assumptions of non-hierarchical algorithms (such as the requirement for apriori knowledge of cluster number of k-means) are not available. Remarkably, the dendrogram

L. Sousa et al. (Eds.): Euro-Par 2021, LNCS 12820, pp. 580–595, 2021.
https://doi.org/10.1007/978-3-030-85665-6_36

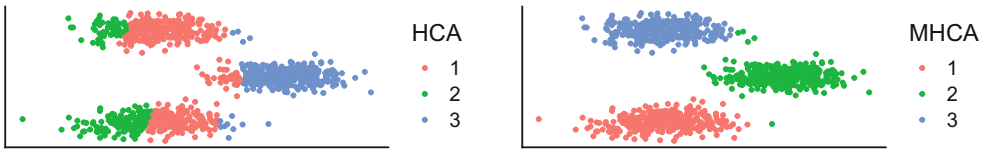


Fig. 1. Mahalanobis-based clustering (MHCA, right) captures the prolonged ellipsoid clusters better than commonly used hierarchical clustering (HCA, left)

output form of HCA provides an ad-hoc dataset ontology which has proven more intuitive for data inspection than the outputs of many other common clustering methods that yield unstructured results.

Here, we focus on hierarchical clustering applications on datasets that originate in flow cytometry, a data acquisition method that allows to quickly measure many biochemical properties of millions of single cells from living organisms. Its widespread use has reached many diverse areas of science including immunology, clinical oncology, marine biology, and developmental biology. The size of the obtained datasets is constantly growing, which naturally drives the demand for fast data processing and advanced analysis methods [11]. From the plethora of developed algorithms, clustering approaches allow easy separation of the measured single cell data into groups that usually correspond to the naturally occurring cell populations and types. Hierarchical clustering improves the result by capturing and revealing more detailed relations between different types of cells.

A dataset from flow cytometry is usually represented as a point cloud in a multidimensional vector space, where each point represents a single measured cell and each dimension represents one measured 'property', typically a presence of some selected surface proteins. Recent hardware development has allowed simple, cheap acquisition of high-quality datasets of several million cells and several dozen of dimensions.

One of the issues in the analysis of this vector space is that the relations between individual dimensions are rather complex, and utilization of simple Euclidean metrics for describing data point similarity is rarely optimal. Fišer et al. [6] have demonstrated the viability of specialized hierarchical clustering analysis method that uses Mahalanobis distance (MHCA) that captures cell clusters of ellipsoid shapes, which are common in cell populations (demonstrated in Fig. 1). Although this approach has proven to detect various elusive dataset phenomena, its scalability remained a concern. In particular, the high computational cost of Mahalanobis distance makes the straightforward implementation on common hardware practically useful only for datasets of up to approximately 10^4 cells.

1.1 Contributions and Outline

In the domain of clustering, algorithm performance has often been successfully improved by proper reimplementation for GPU hardware accelerators [4,8,10]. However, the computation of the MHCA is relatively irregular and rather complex, making the usual acceleration approaches ineffective.

As the main contribution of this paper, we describe our adaptation of MHCA for contemporary GPUs. In particular, we describe a data structure that can be used to accelerate HCA algorithms on GPUs in general, and provide additional insight about efficiency of the specific parts of MHCA algorithm. We subjected the implementation to comprehensive experimental evaluation and compared it with the existing implementation of MHCA to measure the achieved speedup. Finally, we made the implementation available as open-source[1], making it useful for both biological research and further experiments with parallelization of HCAs.

The mathematical and algorithmic overview of MHCA clustering is presented in Sect. 2, Sect. 3 describes our proposed GPU implementation. We summarize the experimental evaluation in Sect. 4. Section 5 puts the our research in proper context with prior work and Sect. 6 concludes the paper.

2 Hierarchical Clustering with Mahalanobis Distance

In this section, we review the necessary formalism and show the Mahalanobis average-linked hierarchical clustering algorithm. The input dataset is a set of points in d-dimensional vector space, here assumed in $\mathbb{R}^d$, which is a common representation for cytometry data [13]. The algorithm produces a binary tree of *clusters* where each resulting cluster is a subset of the input dataset of highly similar ('close' by some metric in the vector space) points.

Mahalanobis distance [12] is defined between a point x and a non-singleton set of compact points P as

$$\delta_M(x, P) = \sqrt{(x - \bar{P})^T (\mathbf{cov}P)^{-1} (x - \bar{P})},$$

where $\bar{P}$ is the centroid (mean) of the set P, and the entries of the covariance matrix are computed as

$$(\mathbf{cov}P)_{ij} = (|P| - 1)^{-1} \cdot \sum_{p \in P} (p_i - \bar{P}_i) \cdot (p_j - \bar{P}_j).$$

One can intuitively view Mahalanobis distance as an Euclidean distance from the cluster centroid that also reflects the shape and the size of the cluster. In particular, in a space that has been linearly transformed so that the covariance matrix of the cluster is a unit matrix, Euclidean and Mahalanobis distance coincide, as shown in Fig. 2.

The MHCA algorithm can be described in steps as follows:

1. *Initialization*: Construct an 'active set' of numbered clusters $P_{1,2,\ldots,n}$, each comprising one input element (data point) as $P_i = \{e_i\}$ for each $i \in \{1 \ldots n\}$ where $\{e_1, \ldots, e_n\}$ denotes the input dataset.
2. *Iteration*: Until the active set contains only a single item, repeat the following:

[1] https://github.com/asmelko/gmhc.

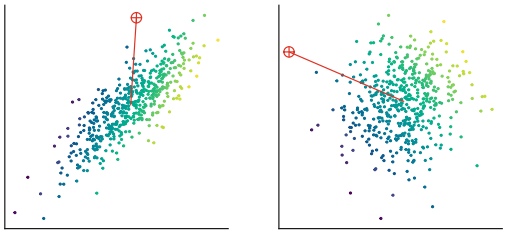


Fig. 2. Mahalanobis distance (left) can be perceived as Euclidean distance (right) in a linearly transformed space where the cluster is perfectly 'round'.

   (a) Compute pairwise dissimilarities of all clusters in A, select the pair (P_r, P_s) with lowest dissimilarity. Output pair (r, s).
   (b) Update the active set by removing P_r, P_s and adding $P_{n+i} = P_r \cup P_s$, where $i > 0$ is an iteration number.
3. *Result*: the binary tree is specified by the trace of $n - 1$ pairs (r, s).

Properties of the output depend mainly on the exact definition of the dissimilarity function used in step 2.a. The common choices include the common 'single' linkage (minimum pairwise distance between the 2 points in different clusters), 'complete' linkage (maximum distance), 'average' linkage (mean distance across clusters), 'centroid' linkage (distance of cluster centroids), and others. The used distance is usually a metric in the vector space, such as Euclidean. The choice of the dissimilarity calculation methods is critical for obtaining results suitable for given analysis; the available methods have been therefore been subjected to much optimization [14].

2.1 Mahalanobis Dissimilarity

Fišer et al. [6] proposed the *full Mahalanobis distance* as a dissimilarity function for HCA as an average of all Mahalanobis distances across clusters, as $\mathrm{FMD}(P_i, P_j) = (|P_i| + |P_j|)^{-1} \left(\sum_k \delta_M((P_i)_k, P_j) + \sum_k \delta_M((P_j)_k, P_i)\right)$. While this construction is intuitively correct and allows the clustering to precisely capture various dataset phenomena that are common in cytometry, the definition opens many inefficiencies and border cases that need to be resolved:

- Mahalanobis distance may be undefined for small clusters because the covariance matrix is singular or nearly-singular. This can be resolved by a complete or partial fallback to robust distance measures, as detailed in Sect. 2.2.
- Because the Mahalanobis distance of a fixed point to a cluster *decreases* when the cluster size increases (e.g., as a result of being merged with another cluster), the minimal dissimilarity selected in the step 3 of the algorithm may sometimes be smaller than the previously selected one. A correction is thus needed to keep the dissimilarity sequence properly monotonic, giving uncluttered, interpretable dendrogram display [5].
- The amount of required computation is significantly higher than with the other linkages (dissimilarity functions), requiring additional operations for

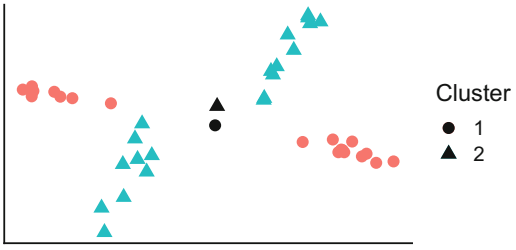


Fig. 3. An example of two clusters for which the CMD fails to satisfactorily approximate the FMD (centroids are plotted in black).

computing the inverted covariance matrix and covariance-scaled Euclidean distances. We mitigate this problem by massive parallelization with GPU accelerators, as detailed in Sect. 3.

The computation of the 'full' average Manalanobis distance is unavoidably demanding, requiring many matrix-vector multiplications to compute distances between all points of one cluster and the opposite cluster. Following the variations of Euclidean dissimilarity measures for HCA, a *centroid-based Mahalanobis distance* may be specified to use only the average of the distance to the centroids of the other cluster, as $\mathrm{CMD}(P_i, P_j) = \frac{1}{2}\left(\delta_M(\bar{P}_i, P_j) + \delta_M(\bar{P}_j, P_i)\right)$. The result may be viewed as a fast approximate substitute for the full variant because the simplification removes a significant portion of the computational overhead and still produces sound results in many cases. The difference between CMD and FMD is highly pronounced only when the centroids of the clusters are near, but their respective covariances differ, as visualized in Fig. 3. Fortunately, such situations are quite rare in clustering of realistic datasets.

2.2 Singularity of Cluster Covariance Matrix

In early iterations of MHCA, the clusters consist of only a few points. Covariance matrix of a small cluster is likely singular, which means it is impossible to compute its inverse required by the Mahalanobis distance measure. Furthermore, even for more points the covariance matrix may be nearly singular, and using its ill-conditioned inverse will yield inaccurate results and numeric floating-point anomalies (such as negative distances or infinities).

To solve this problem, Fišer et al. [6] proposed the following approach: If the number of elements in a cluster relative to whole dataset size is lower than a threshold, the covariance matrix of such cluster is transformed so it can be inverted, and handled in a numerically safe manner. We will denote the used threshold as the *Mahalanobis threshold*, and categorize the clusters as *subthreshold* and *super-threshold cluster*, depending on their size being below and above the Mahalanobis threshold respectively.

We later explore the following *subthreshold handling methods* for managing the problematic covariance matrix values:

- MAHAL smoothly pushes the vectors of the covariance matrices of the sub-threshold clusters towards a unit sphere, so that the space around the clusters is not excessively distorted (or projected).
- EUCLIDMAHAL enforces unit (spherical) covariance vectors of the sub-threshold clusters (thus enforcing Euclidean distances). Despite the simplicity and effectiveness, the hard thresholding may lead to a non-intuitive behavior; for example, the merging of a pair of large elliptical clusters that are just above the threshold may be prioritized over a pair of more similar but sub-threshold clusters.
- EUCLID enforces unit covariances of all clusters *only until the last sub-threshold cluster is merged*. This option usually leads to a viable formation of compact clusters, but completely ignores the possible intrinsic structure of several super-threshold clusters.

2.3 Complexity and Parallelization Opportunities of MHCA

A straightforward serial implementation of MHCA (such as the implementation in `mhca` R package[2]) works with iterative updates of the dissimilarity matrix. Let us examine in detail the time complexity of the individual algorithm steps on a dataset that contains n points of d dimensions:

First, the algorithm constructs a dissimilarity matrix in $\mathcal{O}(d \cdot n^2)$, and identifies the most similar cluster pair in $\mathcal{O}(n^2)$. Then a total of $n-1$ iterations is performed as such:

- a covariance matrix of the merged cluster is computed ($\mathcal{O}(d^2 \cdot n)$) and inverted ($\mathcal{O}(d^3)$),
- the dissimilarity matrix is updated ($\mathcal{O}(d^2 \cdot n)$), and
- the new most similar cluster pair is identified ($\mathcal{O}(n^2)$).

The total complexity is thus $\mathcal{O}(d \cdot n^2 + (n-1) \cdot (d^2 \cdot n + d^3 + n^2))$. Assuming $d \ll n$, the asymptotic complexity can be simplified to $\mathcal{O}(n^3)$. Since we cache the unchanged dissimilarity matrix entries, the memory complexity is $\mathcal{O}(n^2)$.

In an idealized parallel execution environment (PRAM model with concurrent reads and infinite parallelism), we could improve the algorithm to perform faster as follows: All cluster dissimilarity computations (including the later dissimilarity matrix update) can be performed in parallel in $\mathcal{O}(d^3 \cdot \log n)$, using parallel reduction algorithm for computing the covariance sums. The most similar cluster pair can be selected using a parallel reduction over the dissimilarity matrix in $\mathcal{O}(\log^2 n)$. The total required time would thus be reduced to $\mathcal{O}(d^3 n \log^2 n)$ (again assuming $d \ll n$), using $\mathcal{O}(n^2)$ memory.

While this suggests two main ways of performance improvement for the massively parallel GPU implementation, the specifics of the current GPUs pose problems for such naive parallelization approach:

[2] https://rdrr.io/github/tsieger/mhca.

- Parallelization of any single covariance matrix computation will improve performance only if the covariance matrix is sufficiently large, otherwise the performance may be reduced by scheduling overhead and limited parallelism.
- Scanning of the large dissimilarity matrix is parallelizable, but is hindered by relatively small amount of available GPU memory and insufficient memory throughput.

In the following section, we address these problems with optimizations that make the computation viable on the modern accelerators. In particular, we show that the computation of a covariance matrix can be divided into many independent parts, thus exposing sufficient parallelization opportunities, and we demonstrate a technique for efficient caching of intermediate contents of the dissimilarity matrix to reduce the memory footprint and throughput requirements of the algorithm.

3 GPU Implementation

Memory handling optimizations form the essential part of our GPU implementation of MHCA, here called *GMHC* for brevity. Most importantly, we address the tremendous memory requirement of storing the dissimilarity matrix ($\mathcal{O}(n^2)$) for large n. We replace this matrix with a special *nearest-neighbor array*, which provides similar caching benefits, but requires only $\mathcal{O}(n)$ memory. This saving in memory volume is redeemed by a slightly higher computational complexity; however, the measured improvement in scalability warrants this trade-off.

Definition 1 (Nearest neighbor array). *For clusters $P_1, \ldots, P_n$ and a symmetric dissimilarity function d, the* nearest neighbor array N *contains $n-1$ elements defined as*

$$N_i = \text{argmin}_{j>i} d(P_i, P_j).$$

Maintaining a nearest neighbor array in the HCA computation allows us to reduce the amount of distance computations performed after each update. In particular, when a cluster pair (P_i, P_j) is merged into new cluster P_m, only elements with values i and j have to be recomputed, along with the new value for N_m

This is enabled by the symmetry of d, which allowed us to ensure that the contents of the nearest neighbor arrays at some index *only depend on clusters with higher indices*. If we set the new index m to be smaller than all existing indices in the array (i.e., $m = 1$, shifting the rest of the array), the newly appearing cluster can not invalidate the cached indices for the original array, and only the entries that refer to the disappearing clusters i, j need to be recomputed. In consequence, if an already present cluster P_k was to form the most similar pair with the new cluster P_m, this information would be present the $\text{argmin}_{k>m}$ computation, and stored in N_m instead of N_k.

In an optimistic scenario, the above optimization can be used to limit the number of elements that need to be updated in each iteration by a constant

number, which leads to a major increase in overall performance. This constant limit is supported by empirical observations on realistic datasets with around 1 million of objects, where the number of triggered updates was rarely over 50. Further, we reduce the need for recomputation by caching several 'nearest' neighbors for each entry of N:

Definition 2 (Neighbor buffer). *A sorted list of L nearest neighbor indices (respectively to the* $\operatorname{argmin}_{j>i}$ *in Definition 1) stored for each item in N is called a* neighbor buffer.

To ensure the efficiency of the process, we split the update of neighbor buffers to two parts: First, when (P_i, P_j) is merged into P_m, all buffers are filtered and values i and j are removed (i.e., replaced with dummy values). On recomputation, all empty buffers (including newly formed N_m) are filled with indices of nearest L neighbors, while the partially filled buffers are left intact. This allows us to reuse the intermediate results of the computation of an N array entry for as much as L recomputations that involve the cluster.

The complexity of updating the nearest neighbor array element i for the neighbor buffer of size L on m clusters, using a pair dissimilarity computation of complexity $\mathcal{O}(\delta)$, is $\mathcal{O}((m-i)\cdot(\delta+L))$. The reduced amount of index updates thus trades off for index update complexity, depending on L. The optimal choice of L is discussed later in Sect. 4.2.

3.1 Algorithm Overview

The hierarchical clustering of n initial clusters is a series of $n-1$ iterations, such that in each iteration two clusters are merged into one. Before the first iteration, the nearest neighbor array N must be initialized. Each subsequent iteration comprises the following compact steps:

1. Scan the neighbor array and fetch the most similar cluster pair
2. Create a new cluster by merging the cluster pair
   - Compute its corresponding centroid and covariance matrix
   - Transform and invert the covariance matrix
3. Update the neighbor array (only required if $n \geq 3$)

The individual parts of the algorithm may be scheduled and executed dynamically, ordered only the data dependencies as displayed in Fig. 4. Mainly, this allows us to split the update of the neighbor array into update of the neighbors of old clusters (i, j) and the update of the newly created cluster. Naturally, the individual steps are internally implemented as data-parallel operations as well.

In GMHC, we control the iteration loop from the host code, while the work of each update step is implemented within a CUDA kernel. Our code employs CUDA streams [1] to efficiently implement the execution overlaps, creating some high-level task parallelism in the process. In the rest of this section, we detail the implementations of the individual CUDA kernels.

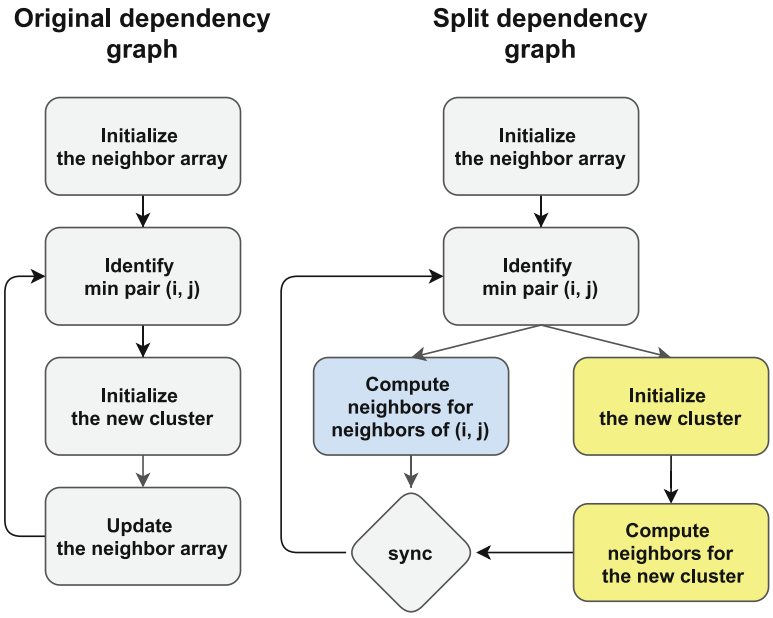


Fig. 4. The original graph of dependencies and the proposed split dependency graph, where blue and yellow boxes can be executed concurrently. (Color figure online)

3.2 Cluster Merging and Covariance Computation

Covariance matrix $\mathbf{cov}P$ of cluster P (and its inversion) is computed only when a cluster is formed; in our case when two clusters are merged. In GMHC, we iterate over all data points $x \in P$ and each item $(\mathbf{cov}P)_{ij}$ is computed as a sum of centered products of x_i and x_j (following the definition from Sect. 2). As the most pressing issue, the performance of this process depends on fast finding of data points x that belong to the cluster in the array of all data points.

A possible straightforward solution, storing an array of assigned points for each data cluster so that the assigned points can be accessed in a fast and compact way, would require dynamic memory allocation or manual apriori over-allocation, and many data moving operations. We settled for a more compact solution with an assignment array that stores a cluster indices for each data point. Although that does not require data copying, both the cluster merge and the retrieval of one cluster points will take $\mathcal{O}(n)$ time. Fortunately, the two operations can be performed by a single parallel scan of the assignment array in this case.

Covariance Kernel Implementation. The covariance kernel takes advantage of the symmetry of a covariance matrix and computes only its upper triangle. Additionally, extra parallelism can be obtained by slicing the computation of the covariance matrix from Sect. 2 over individual data point contributions $\mathbf{S}^x$, as $\mathbf{cov}P = (|P| - 1)^{-1} \sum_{x \in P} \mathbf{S}^x$, where $\mathbf{S}^x_{ij} = (x_i - \bar{\mathbf{x}}_i) \cdot (x_j - \bar{\mathbf{x}}_j)$.

The kernel is implemented as a loop over all data points. A whole CUDA warp is assigned one data point and computes the intermediate $\mathbf{S}^x$. These are then added together in a two-step reduction—all intermediate states within a

CUDA block are reduced using shared memory, which is then followed by a global reduction performed by a separate kernel launch that outputs the totals in a single covariance matrix.

Notably, the covariance matrices of single-point clusters are not computed; rather, they are assigned a default unit matrix.

3.3 Inverse Covariance Storage Optimization

The Mahalanobis distance requires inversion of the covariance matrix, which needs to be computed from the results of the previous step. We use `cuSolver` library[3] for implementing the matrix inversion, namely the routines `potrf` and `potri`.

The inverted matrix is subsequently transformed to better suit the Mahalanobis distance formula, and to eliminate redundant computations later in the process. In particular, we rewrite the Mahalanobis formula for inverse covariance matrix M as a quadratic form

$$x^T M x = \sum_{i=1}^{d} \sum_{j=1}^{d} m_{ij} x_i x_j = \sum_{i=1}^{d} m_{ii} x_i^2 + \sum_{i=1}^{d} \sum_{j>i}^{d} 2 m_{ij} x_i x_j,$$

allowing us to store only the upper-triangular part of the matrix, pre-multiplied by 2.

3.4 Maintenance of Nearest Neighbor Array

GMHC implements 2 similar processes for the neighbor array initialization and update, differing mainly in the granularity of the task size. We thus only focus on the update implementation.

First, specific simplified version of kernel for computing the distances is used for cases when the covariance matrix is unit, falling back to efficient implementation of Euclidean distance. The decision which kernel to execute is done in the host code, depending solely on the selected subthreshold handling method (explained in Sect. 2.2) and the size of the two involved clusters. The decision is formalized in Table 1.

Table 1. The host-side selection of the neighbor-distance kernel

Subthreshold handling method	Sub/sub	Sub/super	Super/super
Euclid	`euclid`	`euclid`	`maha`
EuclidMahal	`euclid`	`maha`	`maha`
Mahal	`maha`	`maha`	`maha`

[3] https://docs.nvidia.com/cuda/cusolver/index.html.

The Neighbor Array Update Kernel. The update operation of nearest neighbor buffer array entry N_i is defined as finding L nearest clusters with index greater than i, and storing their ordered indices into N_i neighbor buffer. We split this operation in two parts, each handled by a separate kernel:

1. Compute distances between all relevant cluster pairs concurrently.
2. Reduce the results into a single nearest neighbor buffer entry.

The execution of the first step differs between the Euclidean and the Mahalanobis neighbor computation. While the former parallelizes trivially with one thread computing one distance value, the complex computation of Mahalanobis distance executes faster if the whole warp cooperates in one distance computation.

The precise operation needed to compute the Mahalanobis distance is a vector-matrix-vector multiplication. To evaluate the formula from Sect. 3.3, we utilize the fuse-multiply-add intrinsic instructions to accumulate the results of the assigned work into their privatized buffers, which are subsequently reduced using fast warp-shuffle instructions.

In the second step, which selects the nearest L indices, is the same for both distance measures. We use a three-level implementation: At the first level, the threads accumulate local minima of small array slices into their registers. At the second level, each thread block utilizes the shared memory to efficiently exchange data and compute block-wise minima. The third level collects the resulting minima and performs the same final reduction on a single thread block (thus efficiently utilizing intra-block synchronization). The second and the third level could be fused together if the atomic instructions were used to synchronize data updates explicitly; however, we observed the improvement was negligible and preferred to reduce the design complexity instead.

This whole neighbor buffer 'refill' operation is performed concurrently for every index in the nearest neighbor array that needs to be updated. Our implementation executes a separate CUDA grid for each update, which reduces implementation complexity but still allows the grids to run concurrently and utilize the entire GPU.

4 Experimental Results

We have subjected our implementation of GMHC to extensive experimental evaluation, measuring the effect of main design choices in the algorithm. In this section, we present the most important results and we put them in proper context, particularly with respect to parameter selection and scaling.

4.1 Benchmarking Methodology and Datasets

The experiments were performed on two systems—a high-end server equipped with NVIDIA Tesla V100 SXM2 (32 GB) and a mainstream PC with NVIDIA

GeForce GTX 980 (4 GB). Both systems used Linux CentOS 8 with CUDA Toolkit (11.2).

We used the original MHCA clustering implementation by Fišer et al. [6] as a baseline, which is, to our best knowledge, the only other publicly available MHCA implementation. The baseline algorithm is written in C as strictly sequential without explicit utilization of SIMD instructions; but it properly utilizes the highly-optimized `Blas` library for most heavy computation. It was benchmarked on a high-end server with Intel Xeon Gold 5218 CPU, clocked at 2.3 GHz (with 64 logical cores) and 384 GB RAM (the same as the high-end server used for benchmarking GMHC). We stress that the comparison between CPU and GPU implementation is not entirely objective, and the test results should be perceived more as a measure of overall data capacity improvement than of the implementation quality. We did not test MHCA on the mainstream PC platform, because of the enormous $\mathcal{O}(n^2)$ memory requirements totaled to hundreds of gigabytes in our benchmarks.

As testing data, we used several high-dimensional datasets originating in mass cytometry [15], namely the `Nilsson_rare` (44K data points, 14 dimensions), `Levine_32dim` (265K data points, 32 dimensions) and `Mosmann_rare` (400k data points, 15 dimensions). For brevity, we report only a subset of the measured results, but these should generalize well to other data. In particular, we did not observe any significant data-dependent performance differences.

In all experiments, we measured the wall time of the total algorithm execution. The experiments were performed multiple times to prevent random deviations in measurement; we display mean values of the measurements. Because the experimental evaluations on both mentioned GPUs behaved consistently with no surprising differences on any particular hardware, we present mainly the results from Tesla V100 SXM2 GPU unless stated otherwise.

4.2 Experiment Results

First, we evaluated the scalability of the GMHC implementation depending on the size of the dataset. The inputs of different sizes were achieved by randomly sub-sampling the Mosmann dataset. Figure 5 shows the wall time for each sub-threshold method, revealing that the performance scales sub-quadratically with data size. Notably, the optimized implementations of Euclid and EuclidMahal scale about 10× better than full Mahal for this dimensionality.

The tradeoff between Euclidean and Mahalanobis computation in the first two methods can be further controlled by setting the threshold value t, controlling whether a cluster is considered small or large, and in turn, deciding the dissimilarity metric to use. Figure 6 summarizes the performance gains for various setting of this threshold.

In the figure, $t = 0$ forces all methods perform dissimilarity measurements using the Mahalanobis distance. When we increase t only very slightly to 0.01, the Euclid method time decreases dramatically and stays almost the same in the remainder of t range. This is often caused by small sub-threshold clusters that are propagated to the very end of the clustering, which postpones the switch to

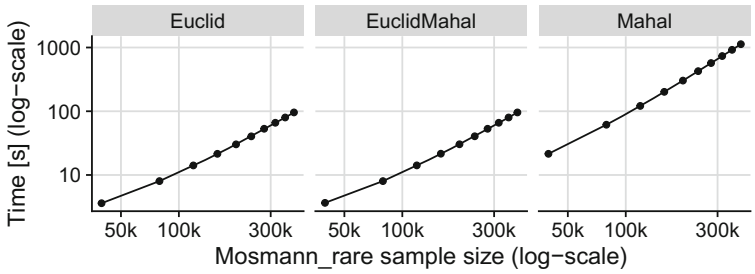


Fig. 5. Comparison of the subthreshold distance computation method performance.

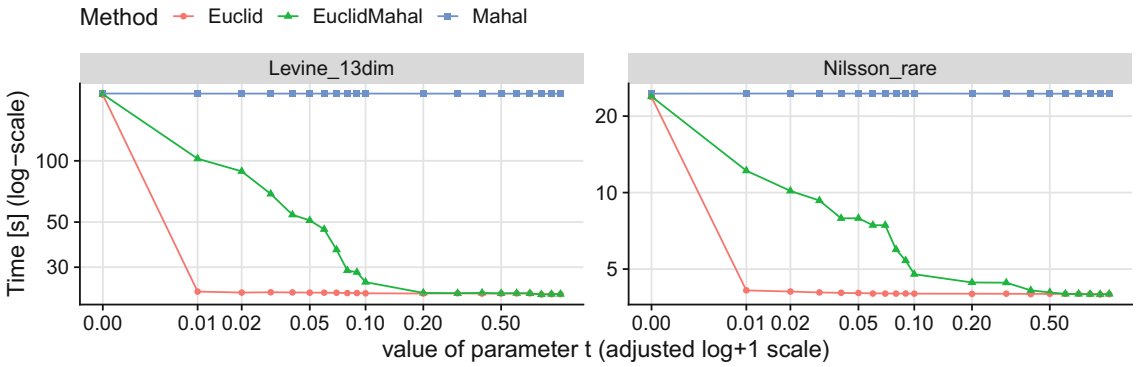


Fig. 6. Clustering time of subthreshold methods with varying Mahalanobis threshold value on two different datasets.

the Mahalanobis distance. On the other hand, the EuclidMahal method shifts its wall time smoothly towards the Euclid method as t increases, which is a consequence of the first super-threshold cluster appearing later in the process.

To determine the optimal value of the nearest neighbor buffer size, we benchmarked the clustering of datasets with a range of parameters L (Fig. 7).

Curiously, the observed results show that while $L = 1$ is optimal for Euclid and EuclidMahal, it performs worst for Mahal method. This is a consequence of the used distance function in dissimilarity measurements—for the Euclid and EuclidMahal method, where the Euclidean distance function dominates, the time difference for performing smaller number of neighbor updates did not balance the increased time complexity of a single update. The Mahal method works optimally with $L = 2$; as the L increases further, the performance starts to decrease again.

Similarly, the optimal value of L increases for higher-dimensional datasets, which we tested on Levine_32dim data (detailed results not shown). In particular, for Mahalanobis distance, we measured the same optimal value $L = 2$ with much greater performance gain (over 30%) against $L = 1$ than on the Mosmann dataset. We expect that the optimal value of L will continue to increase with the dimensionality of the dataset in case of the Mahal method. On the other hand, the Euclidean-based methods kept their optimum at $L = 1$.

Finally, we compared the performance of GPU implementation of MHCA to the CPU baseline, to estimate the outcome for practical data analysis scalability.

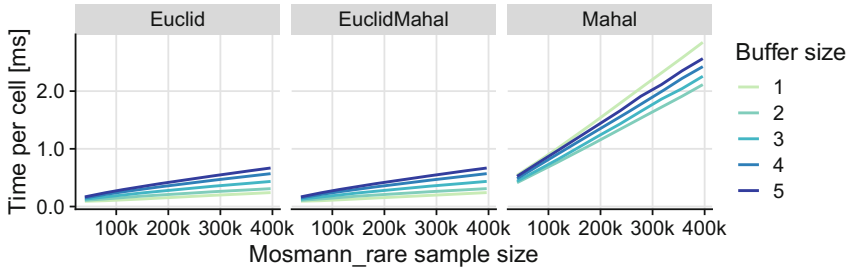


Fig. 7. Comparison of neighbor buffer sizes for subthreshold methods ($t = 0.5$)

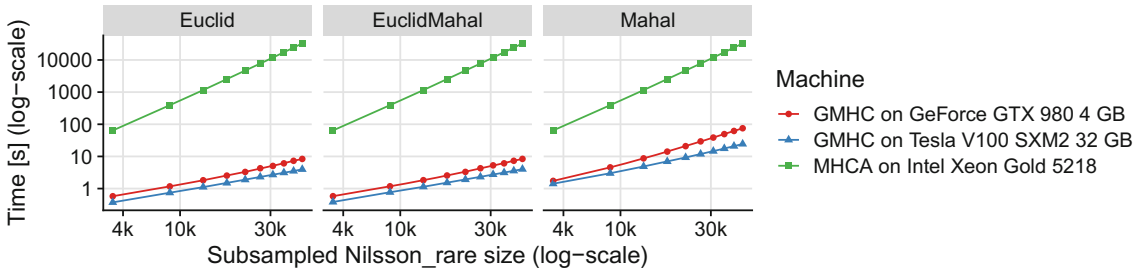


Fig. 8. GMHC and MHCA comparison on Nilsson dataset with default $t = 0.5$.

Figure 8 indicates an overall performance increase by up to 1400× in case of the Mahal method and by up to 8000× in case of mixed-Euclidean methods. When comparing performance on the older 'gaming' GTX 980 GPU, the speedups were around 400× and 4000×, respectively. In summary, modern GPUs have been able to accelerate the MHCA task by more than three orders of magnitude, which is consistent with the effects of parallelization applied to many other clustering algorithms.

5 Related Work

The original version of MHCA clustering for flow cytometry by Fišer et al. [6] used a MATLAB implementation to analyze datasets of around 10^4 multi-dimensional data points. Due to the limited scalability and interoperability with modern data analysis environments, a C version of the algorithm has been implemented within R package `mhca` and enhanced with the possibility of assuming apriori clusters for approximation, to reduce the unfavorable $\mathcal{O}(n^3)$ time complexity for large datasets. That allowed the authors to process datasets of around 10^6 data points within an interactive environments [9].

Despite of the performance advancement, the approximation in the method did not retain the sensitivity required to detect various small clusters of interest (i.e., small cell populations), such as the 'minimum residual disease' cells crucial for diagnosis of acute myeloid leukemia [6]. Similar approximations are used in many other clustering methods to gain performance at the cost of precision justifiable in a specific domain; including the 2-level meta-clustering approach of FlowSOM [7], and advanced approximate neighborhood graph structure of FastPG [2].

Acceleration of HCAs on GPUs has been explored by several authors: Chang et al. [3] discuss hierarchical clustering of gene mRNA levels assayable by DNA microarray technology. Their GPU code computes matrix of pairwise distances between genes using Pearson correlation coefficient as one of the present metrics, and utilized a special property of data to effectively perform single-linkage over the present matrix. Zhang et al. [16] used similar clustering methodology, but employed GPU texture elements for the data representation of gene expression profile HCA. Both acceleration methods resulted in performance increase between 5× to 30× on datasets of 10^4 data points.

6 Conclusions

We have presented an implementation approach for Mahalanobis-average linkage hierarchical clustering algorithm, which utilizes modern parallel GPU accelerators to increase its performance. In the benchmarks, our GPU implementation GMHC has achieved over $10^3\times$ speedup on practical datasets over the current CPU implementations, which enabled scaling of the MHCA algorithm to large datasets produced by current data acquisition methods.

Together with the open-source implementation, we have provided a new high-performance building block for dataset analyses which should support the growing demand for fast data analysis methods not only in cytometry, but also in other areas of data analysis dealing with irregularly shaped Gaussian clusters.

The implementation structure detailed in the paper has allowed us to streamline the utilization of parallel hardware for accelerating general hierarchical clustering algorithms. We expect that the proposed data structures will be ported to support acceleration of dissimilarity measures in other hierarchical clustering methods, providing a solid building block for future acceleration of data mining and knowledge discovery.

Acknowledgements. This work was supported by Czech Science Foundation (GAČR) project 19-22071Y, by ELIXIR CZ LM2018131 (MEYS), by Charles University grant SVV-260451, and by Czech Health Research Council (AZV) [NV18-08-00385].

References

1. CUDA C++ Programming Guide (2021). https://docs.nvidia.com/cuda/cuda-c-programming-guide
2. Bodenheimer, T., et al.: Fastpg: fast clustering of millions of single cells. bioRxiv (2020)
3. Chang, D.J., Kantardzic, M.M., Ouyang, M.: Hierarchical clustering with cuda/gpu. In: ISCA PDCCS, pp. 7–12. Citeseer (2009)
4. Cuomo, S., De Angelis, V., Farina, G., Marcellino, L., Toraldo, G.: A gpu-accelerated parallel k-means algorithm. Comput. Electric. Eng. **75**, 262–274 (2019)
5. Everitt, B., Skrondal, A.: The Cambridge dictionary of statistics, vol. 106. Cambridge University Press, Cambridge (2002)

6. Fišer, K., et al.: Detection and monitoring of normal and leukemic cell populations with hierarchical clustering of flow cytometry data. Cytometry Part A **81**(1), 25–34 (2012)
7. van Gassen, S., et al.: Flowsom: using self-organizing maps for visualization and interpretation of cytometry data. Cytometry Part A **87**(7), 636–645 (2015)
8. Gowanlock, M., Rude, C.M., Blair, D.M., Li, J.D., Pankratius, V.: Clustering throughput optimization on the gpu. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 832–841. IEEE (2017)
9. Kratochvíl, M., Bednárek, D., Sieger, T., Fišer, K., Vondrášek, J.: Shinysom: graphical som-based analysis of single-cell cytometry data. Bioinformatics **36**(10), 3288–3289 (2020)
10. Kruliš, M., Kratochvíl, M.: Detailed analysis and optimization of cuda k-means algorithm. In: 49th International Conference on Parallel Processing-ICPP, pp. 1–11 (2020)
11. Lugli, E., Roederer, M., Cossarizza, A.: Data analysis in flow cytometry: the future just started. Cytometry Part A **77**(7), 705–713 (2010)
12. Mahalanobis, P.C.: On the generalized distance in statistics. National Institute of Science of India (1936)
13. Shapiro, H.M.: Practical Flow Cytometry. John Wiley & Sons, Hoboken (2005)
14. Shirkhorshidi, A.S., Aghabozorgi, S., Wah, T.Y.: A comparison study on similarity and dissimilarity measures in clustering continuous data. PloS one **10**(12), e0144059 (2015)
15. Weber, L.M., Robinson, M.D.: Comparison of clustering methods for high-dimensional single-cell flow and mass cytometry data. Cytometry Part A **89**(12), 1084–1096 (2016)
16. Zhang, Q., Zhang, Y.: Hierarchical clustering of gene expression profiles with graphics hardware acceleration. Pattern Recogn. Lett. **27**(6), 676–681 (2006)

Contribution 2
EmbedSOM

Published as Adam Šmelko et al. "GPU-acceleration of neighborhood-based dimensionality reduction algorithm EmbedSOM". in: *16th Workshop on General Purpose Processing Using GPU*. Association for Computing Machinery, 2024, pp. 13–18

GPU-acceleration of neighborhood-based dimensionality reduction algorithm EmbedSOM

Adam Šmelko
Martin Kruliš
Jiří Klepl
smelko@d3s.mff.cuni.cz
krulis@d3s.mff.cuni.cz
klepl@d3s.mff.cuni.cz
Department of Distributed and Dependable Systems, Charles University
Prague, Czechia

ABSTRACT

Dimensionality reduction methods have found vast applications as visualization tools in diverse areas of science. Although many different methods exist, their performance is often insufficient for providing quick insight into many contemporary datasets. In this paper, we propose a highly optimized GPU implementation of EmbedSOM, a dimensionality reduction algorithm based on self-organizing maps. We detail the optimizations of k-NN search and 2D projection kernels which comprise the core of the algorithm. To tackle the thread divergence and low arithmetic intensity, we use a modified bitonic sort for k-NN search and a projection kernel that utilizes vector loads and register caches. The evaluated performance benchmarks indicate that the optimized EmbedSOM implementation is capable of projecting over 30 million individual data points per second.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**; **Designing software**.

KEYWORDS

Dimensionality reduction, Single-cell cytometry, GPU acceleration, CUDA, kNN, Optimizations

ACM Reference Format:
Adam Šmelko, Martin Kruliš, and Jiří Klepl. 2024. GPU-acceleration of neighborhood-based dimensionality reduction algorithm EmbedSOM. In *16th Workshop on General Purpose Processing Using GPU (GPGPU '24), March 02, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649411.3649414

1 INTRODUCTION

Dimensionality reduction algorithms emerged as indispensable utilities that enable various forms of intuitive data visualization, providing insight that in turn simplifies rigorous data analysis. The development has benefited especially the life sciences, where algorithms like t-SNE [16] reshaped the accepted ways of interpreting many kinds of measurements, such as genes, single-cell phenotypes and development pathways, and behavioral patterns [2, 15].

The performance of the non-linear dimensionality reduction algorithms becomes a concern if the analysis pipeline is required to scale or when the results are required in a limited amount of time such as in clinical settings. To tackle the limitations of poor scalability, Kratochvíl et al. developed EmbedSOM [7], a dimensionality reduction and visualization algorithm based on self-organizing maps (SOMs) [5]. EmbedSOM provided a 10× speedup on datasets typical for single-cell cytometry data visualization while retaining the competitive quality of the results. Still, the parallelization potential of EmbedSOM remained mostly untapped as of yet.

This paper describes an efficient, highly parallel GPU implementation of EmbedSOM designed to provide real-time results on large datasets. The implementation is accompanied by performance benchmarks of individual optimizations to evaluate the optimal variants for different dataset sizes. Both the implementation and the data are available in our GitHub repository[1]. Furthermore, the repository also contains the figures that were omitted due to space constraints.

In the paper, we first describe the EmbedSOM algorithm in Section 2. We specifically detail the CUDA-based GPU implementation of the algorithm in Section 3 and evaluate its performance in Section 4. Related work is discussed in Section 5 and Section 6 concludes the paper.

2 LANDMARK-DIRECTED REDUCTION

EmbedSOM is a visualization-oriented method of non-linear dimensionality reduction that works by describing high-dimensional points by their locations relative to landmarks equipped with a topology and reproducing the point in a low-dimensional space using an explicit low-dimensional projection of the landmarks with the same topology [7].

Formally, the EmbedSOM algorithm works as follows. Let d be the dimension of the high-dimensional space and assume $\mathbb{R}^2$ is the low-dimensional space for brevity. EmbedSOM processes n d-dimensional points in a matrix X of size $n \times d$, and outputs n 2-dimensional points in a matrix x of size $n \times 2$. The high- and low-dimensional landmarks similarly form matrices L of size $g \times d$ and

[1] https://github.com/asmelko/gpgpu24-artifact

GPGPU '24, March 02, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-1817-5/24/03
https://doi.org/10.1145/3649411.3649414

l of size $g \times 2$, where usually $g \ll n$. Each point X_i is transformed to a point x_i as:

(1) k nearest landmarks are found for point X_i (k is a constant parameter satisfying $3 \leq k \leq g$)
(2) the landmarks are ordered and scored by a smooth distance function that assigns the highest score to the closest landmark and 0 to the k-th landmark (ensuring the smoothness of projection in cases $k < g$ [7])
(3) for each pair (u, v) of the closest $k-1$ landmarks (the ones with non-zero score), a projection of the point X_i is found on the 1-dimensional affine space with coordinate 0 at L_u and 1 at L_v; the 1-dimensional coordinate of the projection in this affine space is taken as $D_{uv}(X_i)$ and the same projected coordinates are defined in the low-dimensional space as $d_{uv}(x_i)$
(4) point x_i is fitted to the low-dimensional space so that the squared error in the coordinates weighed by nearest landmark scores (s_u, s_v) is minimized:

$$x_i = \arg\min_{p \in \mathbb{R}^2} \sum_{u,v} s_u \cdot s_v \cdot (D_{uv}(X_i) - d_{uv}(p))^2$$

Because $d_{uv}(p)$ is designed as a linear operator, the error minimization problem (step 4) collapses to a trivial solution of 2 linear equations with 2 variables. A complete algorithm may be found in the original publication [7, Algorithm 1].

3 GPU IMPLEMENTATION OF EMBEDSOM

While EmbedSOM is relatively straightforward to parallelize for mainstream CPU architectures, several challenges appear when optimizing for contemporary GPUs. This section outlines the key optimizations that made the high-performance EmbedSOM implementation possible and overviews the relative performance gains achieved by individual choices made. The algorithm consists of two main parts to address which we describe in the remainder of the section:

- **k-NN step** The search of k-nearest landmarks in L for each data point from X requires a highly irregular selection of indices of k lowest values from columns of the dynamically computed distance matrix $L^T \cdot X$.
- **Projection step** Computation of the small linear system that is used to find the projection of a point, namely of projections D_{uv} and the derivatives $\frac{\delta d_{uv}}{\delta x_i}$, is difficult to optimize due to irregular memory access patterns of collecting the data for the computation.

3.1 k-NN selection step

The task of the first part of the algorithm is to find k nearest landmarks (from L) for every data point in X. This comprises two substeps: computing Euclidean distances for every pair from L and X and performing point-wise reduction that selects a set of k nearest landmarks for each of the n points, based on the computed distances.

While the Euclidean distance computation is mathematically simple and embarrassingly parallel, achieving optimal throughput on GPUs is quite challenging [10]. In particular, the ratio between the data transfers and the arithmetic operations performed by each GPU core is heavily biased towards data transfers. The overhead of data transfers is best prevented by finding a good caching pattern for the input data that is able to optimally utilize all hardware caches (L1 and L2), shared memory, and core registers.

The parallel implementation of the k-NN search is even more challenging. The k-NN problem is computed individually for each data point, which provides the space for possible parallelization. However, concurrently processed instances of a naïve k-NN implementation exhibit severe code divergence because the selection process is purely data-driven, and requires a high amount of memory allocated per core. Optimally, the k-NN selection is realized by customized versions of parallel sorting algorithms, which are well-researched and possess existing GPU implementations [13].

Our implementation chooses to optimize both sub-steps since the ratio of the amount of required computations can be easily biased by the configuration of parameters d and k. In particular, processing high-dimensional datasets with a low k parameter spends significantly more time in the distance computation, but lower-dimensional datasets with higher k require more time in the nearest neighbor selection.

Concerning the perspective of software design, the implementation may use separate kernels for both sub-tasks or a single fused kernel. Kernel separation provides better code modularity and more flexibility in work-to-thread division and data caching strategy, at the cost of having to materialize all the computed distances in the GPU global memory, thus significantly increasing the total amount of data transfers. In contrast to that, a fused kernel may immediately utilize the computed distances in k-NN computation without transferring the data to global memory and interleaving the distance computations with k-NN may help to improve the ratio between computations and data transfers. Since our initial observations showed that the overhead of the data transfers required for kernel communication is relatively high, we decided to implement only the fused variant for the sake of simplicity. The usage of separate kernels might be interesting in the future, especially for extreme values of d that diminish the relative cost of the distance data transfer.

3.1.1 Available algorithms for k-NN. There are many approaches to k-NN selection, varying in complexity and parameter-dependent performance. We implemented several of the possibilities (as described in this section) to substantiate our choice of the algorithm for GPU EmbedSOM.

As a baseline (labeled **Base**), we used the most straightforward approach to GPU parallelization which simply invokes original sequential code for every data point concurrently. The Base kernel is spawned in n threads (one for each data point), and each thread computes the distance between its data point and all landmarks while maintaining an ordered array of k nearest neighbors. The array is updated by an insert-sort step performed for every new computed distance — i.e., by starting at the end of the array and moving the new distance-index pair towards smaller values until it reaches the correct position.

Shared algorithm is a modified version of the baseline algorithm that utilizes shared memory as a cache, following the recommended optimization practice of improving performance by caching data that are reused multiple times [12]. In this case, we

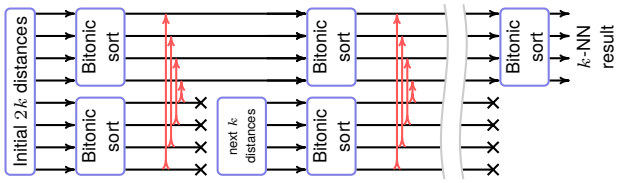


Figure 1: Bitonic algorithm for k-NN selection ($k = 4$). Each horizontal line represents a data item in the shared memory. Red lines represent comparators ensuring, that the intermediate k 'best' neighbors and in the top buffer.

cache the landmark coordinates, which are sufficiently small to fit in the shared memory for all tested parametrizations.

In **GridInsert** algorithm, we utilize the shared memory to cache both landmarks and points. However, the limited size of shared memory imposes limitations of the amount of cached data. Hence, the algorithm was parametrized by the block height h (number of cached points from X) and the block width w (number of cached landmarks from L). The algorithm runs in epochs, each of which first caches h points and w landmarks, and then computes $h \cdot w$ distance values using only data in shared memory. While the distances are computed concurrently by the whole thread block, we chose to avoid explicit synchronization in the k-NN step, using only h threads to incorporate the newly computed distances into h separate k-NN results using the insert-sort steps. The GridInsert should achieve better throughput in the distance computation thanks to the caching, at the cost of slightly sub-optimal k-NN reduction; thus, giving the best performance on high-dimensional datasets and low values of k.

Finally, improvising on our previous work [8], we implemented **Bitonic** k-NN selection algorithm, which utilizes routines from the highly parallelizable bitonic sorting algorithm. Bitonic sorting is very suitable for parallel lockstep execution [10], and the capability to merge sorted sequences has allowed us to keep only $2k$ distances (instead of g) in the shared memory. This method benchmarked the best on the average, so it is selected as default for EmbedSOM and we describe it more thoroughly in the following.

3.1.2 Bitonic approach to k-NN. The Bitonic approach can be seen as a combination of the benefits of the other algorithms: It does not require materializing all distances in the memory to do a full sort and even though it does not use an elaborate input caching strategy like GridInsert, it still gives interesting results because the data loading operations can be partially overlapped with bitonic sorting operations if enough warps are allocated to one streaming multiprocessor.

The bitonic comparator network provides a building block that, given two buffers of size k of neighbor distances sorted by bitonic sort, selects the closest k of the neighbors in a single (parallel) operation, allowing us to quickly discard neighbors that do not belong into the k-neighborhood. Applying this operation iteratively on k-sized blocks of distances sorted by the bitonic sort (as shown in Figure 1), we obtain a highly performing scheme that requires only $2k$ items present in the shared memory. In particular, the shared memory always contains a k-block of distances (and corresponding indexes) that holds k so-far-nearest neighbors, and one block of k distances that are computed from L; in each iteration, both blocks are sorted by the bitonic sorter in parallel and merged by the bitonic comparator to move the distances of new nearest k neighbors into the intermediate block. The other block is then re-filled by a new set of k distances from L.

Technically, each step of the sorting net requires $\frac{k}{2}$ comparators, thus optimally $\frac{k}{2}$ threads that work concurrently on the h-sized block. Hence, we allocate k threads for each data point, which alternate their work between computing a block of k distances and performing two bitonic sorts on two k-sized blocks in parallel. For simplicity, our implementation assumes that k is always a power of 2, and excessive output of the sorter is discarded.

3.2 Projection step

The second part of the dimensionality reduction method is the actual projection into the low-dimensional space. The computation of the low-dimensional point position x_i by EmbedSOM involves: (1) Conversion of the distances collected in the k-NN to scores; (2) Orthogonal projection of X_i to $\binom{k}{2}$ lines generated by the k neighbors to create contributions to the final approximation matrix; (3) Solution of the resulting small linear system using Cramer's rule.

Since the first and the last steps are embarrassingly parallel problems with straightforward optimal implementation and since the second step is the most time demanding (performing $O(k^2)$ operations on vectors of size d), we focus mainly on the orthogonal projections. Its computation is complicated by a highly irregular pattern of repeated accesses to an arbitrary k-size subset of L. We designed several algorithms that successively optimize the access patterns, detailed below.

The baseline algorithm **Base** uses the most straightforward parallel approach (similar to Base k-NN), where each thread computes the projection of one single point sequentially so the concurrency is achieved only by processing multiple points simultaneously. All data are stored in the global memory, and no explicit cache control is performed.

The irregular repeated access to the elements of L hinders the performance of the baseline algorithm. In the **Shared** algorithm, we chose to reorganize the workload so that each projection is computed by a whole block of threads that cooperatively iterate over the landmark pairs. As a result, the input data of the orthogonal projection (i.e., the k nearest neighbors from L together with the distances, scores, and 2D versions of the landmarks) can be cached in shared memory. The intermediate sub-results represented by 2×3 matrices are successively added into privatized copies of each thread to avoid explicit synchronization and aggregated at the end using a standard parallel reduction, enhanced with warp-shuffle instructions (a similar scheme is used in optimal CUDA k-means implementation [9]).

Because the data transfers comprise a considerable portion of the Shared algorithm execution time, we have optimized the transfers using alignment and data packing techniques, yielding the **Aligned** algorithm. The implementation is based on using vector data types (e.g. `float4` in CUDA) to enable utilization of 128-bit load/store instructions, which improves overall data throughput.

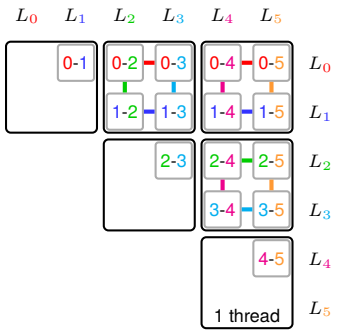


Figure 2: Detail of the caching of landmark data in REGISTERS projection kernel. Multiple landmark pairs (small boxes) are processed by each thread (large boxes). Caching of the landmark data in registers allows the reuse of loaded data (color lines), thus reducing the amount of memory accesses.

The vectorization comes only at a relatively small cost of aligning and padding the vectors to 16-byte blocks.

To further improve the data caching, we implemented algorithm REGISTERS, where each thread computes more than one landmark pair in a single iteration so that the coordinates loaded into its registers can be shared as inputs among multiple landmark-pairs computations. The data sharing scheme is detailed in Figure 2. We found that it is optimal to group the threads into small blocks of 2×2 computation items, saving half of the data loads. Larger groups are theoretically possible, but even 3×3 caused excessive registry pressure and impaired performance on contemporary GPUs. The innermost loop of the algorithm iterates over d so that only a single `float4` value per each landmark is kept in registers.

4 EXPERIMENTAL RESULTS

The main objective of the benchmarking was to measure the speedups achieved by different applied optimizations and to determine the optimal algorithms and their parameter setting for the sub-tasks of EmbedSOM computation.

The timing results, presented in the following sections, were collected as kernel execution times measured by a standard system high-precision clock. Each test was repeated 10× and the mean values are presented in the subsequent figures. The relative standard deviations of the measurements were less than 5% so we chose not to include them. Complete measurements are available in our GitHub repository[2].

Results were collected on NVIDIA Tesla A100 PCIe 80 GB running CUDA 12.2. All benchmarking datasets were synthetic, containing exactly 1Mi points ($n = 2^{20}$, reflecting the common sizing of real-world datasets [1]) with all coordinates sampled randomly from the same uniform distribution. The performance of the benchmarked algorithms is not data-dependent, except for the case of caching performance in the projection step, where the completely random dataset is the worst-case scenario.

[2] https://github.com/asmelko/gpgpu24-artifact

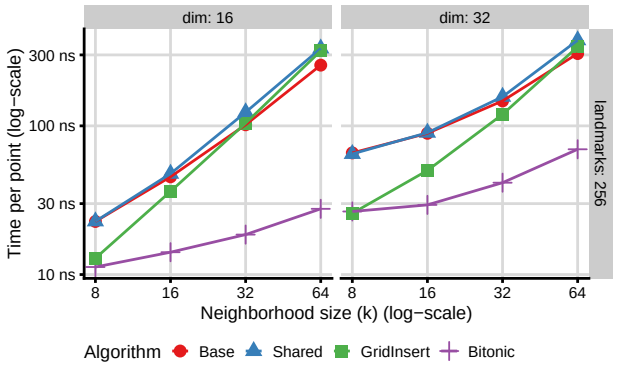


Figure 3: Amortized performance of k-NN step for a single input point using parameters usual in flow cytometry

4.1 Performance of k-NN selection

Here we give an overview of performance and viable parameter settings observed for the k-NN selection algorithms.

Notably, all algorithms for k-NN are affected by CUDA thread block sizing which affects warp scheduling and data reuse possibilities of the shared-memory cache. We observed that the total thread block size of 256 threads was either optimal or near to optimal for almost all tested configurations, except for GRIDINSERT that performed the best with 64 threads for lower values of d and g parameters.

Parameters w and h of the GRIDINSERT algorithm determine the ratio between data transfers and computations, but may also affect the pressure on the shared memory[3]. Empirical evaluation indicates that the algorithm performs the best when each parallel insertion sort is performed in a separate warp, so the code divergence in SIMT execution is prevented (i.e., w is a multiple of 32) The optimal performance was observed for w equal to 96 or 128; However, the speedup over $w = 32$ is relatively low.

A comparison of the best parametrizations of each algorithm on various configurations common in our target use cases is shown in Figure 3. The BITONIC algorithm significantly outperformed the other algorithms. The speedup of BITONIC over BASE was between 3× to 20× and usually more than 2× over the second-ranking method.

The benchmarking also confirmed a rather huge scaling difference between algorithms based on divergent insertion sort and algorithms based on sub-quadratic parallelizable sorting schemes. We conclude that despite the simplicity that might enable GPU speedups in certain situations, the insertion sort is too slow for larger values of k in this case.

As an interesting result, we observed that despite following the general recommendations, the straightforward use of shared memory (in the SHARED algorithm) did not improve overall performance over the BASE. Quite conversely, the overhead of explicit caching even caused a slight decrease in the overall performance.

We additionally report the performance measurements for two selected corner cases with extreme values of g and d (figure omitted

[3] Technically, parameter h is determined by the thread block size divided by w, we thus optimize only w.

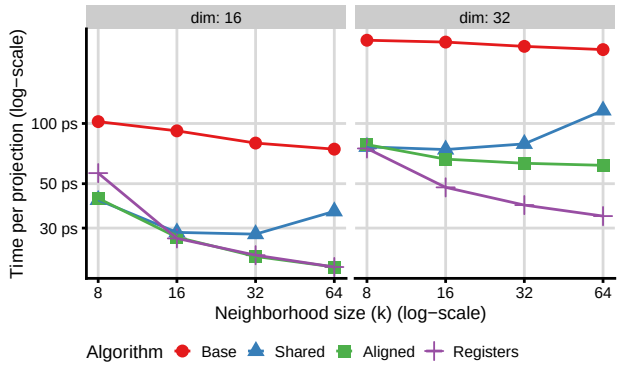


Figure 4: Amortized performance of a single projection operation in the algorithms that compute the projection step (showing the most important problem parametrizations)

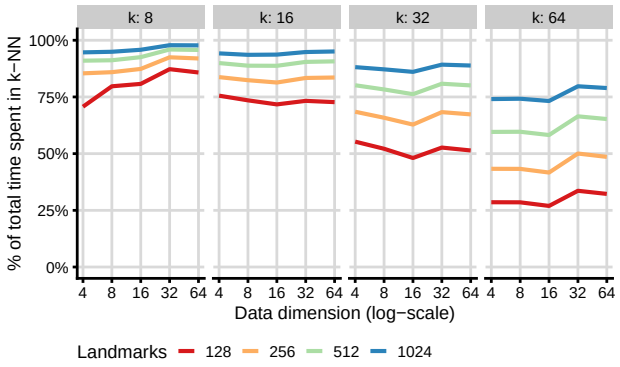


Figure 5: The relative time spent by the k-NN computation usually dominates the execution of GPU EmbedSOM, composed of Bitonic+Registers algorithms. Projection computation time becomes dominant only for relatively impractical parametrizations of low g and high k.

due to the page limit). Mainly, the total volume of the computation required to prepare the Euclidean distances scales with $g \cdot d$, which becomes dominant when both are maximized. At that point, we observed that GridInsert provides comparable or mildly better performance than Bitonic, especially in cases where k is small and the overhead of insertion sorting is not as pronounced.

Naturally, we should ask whether it could be feasible to combine the benchmarked benefits of GridInsert and Bitonic algorithms in order to get the best of both approaches (optimal inputs caching and fast k-NN filtering). While an investigation of this possibility could be intriguing, we observed that a fused algorithm would require very complicated management of the shared memory (which both algorithms utilize heavily), and the estimated improvement of performance was not sufficient to substantiate this overhead; we thus left the question open for future research.

4.2 Performance of projection step

The projection algorithms described in the previous section have only two execution parameters: The size of the CUDA thread block and the number of data points assigned to a thread block (threads are divided among the points evenly). We observed that selecting more than one point per thread block is beneficial only in the case of relatively small problem instances (low k and d) because it prevents underutilization of the cores.

The optimal size of the CUDA thread blocks depends mainly on the parameters k and d. In case of Shared algorithm, optimal values ranged from 32 (for $k = 8$, $d = 4$) to 64 ($k = d = 64$). With the caching optimizations in Aligned and Registers, the optimal thread block size was slightly higher, reaching 128 for the most complex problem instances. We assume this is a direct consequence of the improved memory access efficiency which gives space for parallel execution of additional arithmetic operations.

Figure 4 shows the performance of the best algorithm configurations for the representative parametrizations. All three algorithms perform almost equally for small k, giving around 3× speedup over Base. The importance of optimizations in Aligned and Registers grows steadily when parameter k increases, up to around 10× speedup at $k = 64$. In conclusion, the optimal algorithm for the EmbedSOM projection is determined by the dimensionality of the dataset — Registers performs better at higher dimensions ($d \geq 32$) while Aligned was slightly better for lower dimensions.

4.3 Complete algorithm

A complete GPU implementation of the EmbedSOM algorithm is the combination of the best implementations of k-NN and projection steps. The selected algorithms Bitonic and Registers are simply executed sequentially on large blocks of X, sharing only a single data exchange buffer for transferring the k-NN data. Notably, since the data exchange between the algorithm parts is minimal, comprising only distances and neighbor indexes from the k-NN selection, we claim that no specific optimizations of the interface are required. Our implementation can provide a speedup between 200× and 1000× over a serial CPU implementation, and between 3× and 10× over a straightforward GPU implementation that we used as a baseline (figure omitted due to the page limit).

Finally, we highlight the relative computation complexity of both steps (Figure 5), which changes dynamically with k and might be viable as a guide for further optimization. We observed that for common parametrizations ($k \simeq 20$, $g \simeq 500$), most of the computation time is spent in k-NN step, and projection performance becomes problematic only in cases of almost impractically high k. The results align with the asymptotic time complexities of the algorithms, roughly following $O(n \cdot d \cdot g \cdot \log_2 k)$ for the k-NN and $O(n \cdot d \cdot k^2)$ for the projection.

5 RELATED WORK

The essential component of our success is GPU acceleration of the projection computation which needs to be fast enough to recalculate the embedding in real-time. In the following, we address the most relevant works that influenced or inspired our solution.

Being one of the most profound visualization methods, *t-SNE* was studied to explore the possibilities of having a fast GPU-enabled implementation. One of the initial implementations was t-SNE-CUDA library [3]. The most complicated step (computing the attractive forces of the N-body simulation) is handled as a multiplication of

a sparse matrix and a vector by the CUSparse library. This work was slightly improved a year later [4] when the authors replaced the CUSparse library with their implementation of multiplication, which takes advantage of atomic operations to perform the reduction in scalar sums.

Perhaps the most popular contemporary method for data visualization is the *Uniform Manifold Approximation and Projection* algorithm (UMAP), which often produces better results than t-SNE at the cost of higher computational demands. There are two GPU implementations worth mentioning which were both made part of RAPIDS cuML library [11, 14]. They both use a similar approach, implementing a k-NN approximation based on gradient descent methods. The first implementation [14] relies more on existing solutions and libraries, and the second one [11] is slightly more low-level as they implement the embedding using custom kernels.

Even though the presented methods (especially t-SNE) exceeded the speedup of two orders of magnitude, they are still quite far from real-time processing when the number of points reaches the order of millions. The proposed EmbedSOM projection is based on SOMs and linear projection based on k-NN search [6, 7], which is technically closest to the work of Yeh et al. [17]. For the SOM part, we have adapted the state-of-the-art implementation of k-means algorithm [9] since SOM shares many of its steps. The crucial part of the projection is the k-NN search, which is also repeated in the aforementioned papers; however, we have found that the solution based on bitonic-sorting [8] performs the best in our case.

6 CONCLUSION

We have presented a GPU implementation for the semi-supervised dimensionality reduction algorithm EmbedSOM where we optimized independently two kernels: A general kNN search and a 2D projection which may be used independently. The k-NN was solved by adapted bitonic sorting, which eliminates thread divergence. The projection kernel was optimized to fetch and use data most efficiently by utilizing vector loads and data reuse on the register level. Thorough benchmarking indicated that both kernels achieved a significant speedup over the baseline GPU implementation.

The proposed implementation should enable subsequent research in interactive dimensionality reduction tools where the user changes SOM parameters or landmarks and the projections are re-computed and visualized in real-time. The results show that the optimized EmbedSOM version can project more than 1 million individual data points each frame, while maintaining a frame rate above 30fps.

ACKNOWLEDGMENTS

This paper was supported by Charles University institutional funding SVV 260698 and Charles University Grant Agency (GAUK) project 269723.

REFERENCES

[1] Aysun Adan, Günel Alizada, Yağmur Kiraz, Yusuf Baran, and Ayten Nalbant. 2017. Flow cytometry: basic principles and applications. *Critical reviews in biotechnology* 37, 2 (2017), 163–176.

[2] Jessica Cande, Shigehiro Namiki, Jirui Qiu, Wyatt Korff, Gwyneth M Card, Joshua W Shaevitz, David L Stern, and Gordon J Berman. 2018. Optogenetic dissection of descending behavioral control in Drosophila. *Elife* 7 (2018), e34275.

[3] David M Chan, Roshan Rao, Forrest Huang, and John F Canny. 2018. T-SNE-CUDA: GPU-Accelerated T-SNE and its Applications to Modern Data. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE Computer Society, 330–338.

[4] David M Chan, Roshan Rao, Forrest Huang, and John F Canny. 2019. GPU accelerated t-distributed stochastic neighbor embedding. *J. Parallel and Distrib. Comput.* 131 (2019), 1–13.

[5] Teuvo Kohonen. 1990. The self-organizing map. *Proc. IEEE* 78, 9 (1990), 1464–1480.

[6] Miroslav Kratochvíl, David Bednárek, Tomáš Sieger, Karel Fišer, and Jiří Vondrášek. 2020. ShinySOM: graphical SOM-based analysis of single-cell cytometry data. *Bioinformatics* 36, 10 (2020), 3288–3289.

[7] Miroslav Kratochvíl, Abhishek Koladiya, and Jiří Vondrášek. 2019. Generalized EmbedSOM on quadtree-structured self-organizing maps. *F1000Research* 8, 2120 (2019), 2120. [version 2; peer review: 2 approved].

[8] Martin Kruliš, Hasmik Osipyan, and Stéphane Marchand-Maillet. 2015. Optimizing sorting and top-k selection steps in permutation based indexing on gpus. In *East European Conference on Advances in Databases and Information Systems*. Springer, 305–317.

[9] Martin Kruliš and Miroslav Kratochvíl. 2020. Detailed Analysis and Optimization of CUDA K-means Algorithm. In *49th International Conference on Parallel Processing (ICPP '20)*. ACM.

[10] Martin Kruliš, Hasmik Osipyan, and Stéphane Marchand-Maillet. 2017. Employing GPU architectures for permutation-based indexing. *Multimedia Tools and Applications* 76, 9 (2017), 11859–11887.

[11] Corey J Nolet, Victor Lafargue, Edward Raff, Thejaswi Nanditale, Tim Oates, John Zedlewski, and Joshua Patterson. 2020. Bringing UMAP Closer to the Speed of Light with GPU Acceleration. *arXiv preprint arXiv:2008.00325* (2020).

[12] NVIDIA. 2013. CUDA C Best Practices Guide.

[13] Dhirendra Pratap Singh, Ishan Joshi, and Jaytrilok Choudhary. 2018. Survey of GPU based sorting algorithms. *International Journal of Parallel Programming* 46, 6 (2018), 1017–1034.

[14] Yezihalem Tegegne, Zhonglin Qu, Yu Qian, and Quang Vinh Nguyen. 2021. Parallel Nonlinear Dimensionality Reduction Using GPU Acceleration. In *Australasian Conference on Data Mining*. Springer, 3–15.

[15] Shadi Toghi Eshghi, Amelia Au-Yeung, Chikara Takahashi, Christopher R Bolen, Maclean N Nyachienga, Sean P Lear, Cherie Green, W Rodney Mathews, and William E O'Gorman. 2019. Quantitative comparison of conventional and t-SNE-guided gating analyses. *Frontiers in immunology* 10 (2019), 1194.

[16] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[17] Tsung Tai Yeh, Tseng-Yi Chen, Yen-Chiu Chen, and Wei-Kuan Shih. 2010. Efficient parallel algorithm for nonlinear dimensionality reduction on GPU. In *2010 IEEE International Conference on Granular Computing*. IEEE, 592–597.

Contribution 3
Cross-Correlation

Yet to be published.

Efficient GPU-accelerated Parallel Cross-correlation

Karel Maděra, Adam Šmelko, Martin Kruliš

[a] *Department of Distributed and Dependable Systems, Charles University, Prague, Czech Republic*

Abstract

Cross-correlation is a data analysis method widely employed in various signal processing and similarity-search applications. Our objective is to design a highly optimized GPU-accelerated implementation that would speed up the applications and also improve energy efficiency since GPUs could be more efficient than regular CPUs. There are two rudimentary ways to compute cross-correlation — a definition-based algorithm that tries all possible overlaps and an algorithm based on the Fourier transform, which is much more complex but has better asymptotical time complexity. We have focused mainly on the definition-based approach which is better suited for smaller input data and we have implemented multiple CUDA-enabled algorithms with multiple optimization options. The algorithms were evaluated on various scenarios, including the most typical types of multi-signal correlations, and we provide empirically verified optimal solutions for each of the studied scenarios.

Keywords: cross-correlation, GPU, CUDA, parallel, algorithm, caching, optimizations

1. Introduction

Signal processing and analysis are essential in a plethora of applications ranging from computer vision [1], acoustic localization [2], or processing sensory inputs in various domains in astronomy [3], geology [4], biology [5], or medicine [6]. Cross-correlation is one of the basic methods employed in signal processing since it provides a metric that compares two signals and allows us to detect the best overlap including the relative shift between two signals.

In this work, we focus solely on the efficiency of the cross-correlation algorithm implementation and we aim to design optimizations that should speed up the computation. We tackle the problem with parallel computing, namely employing contemporary GPU accelerators which are particularly suited for data-parallel tasks. Although the task seems simple at first glance, achieving optimal efficiency is quite challenging due to the unique lock-step execution model of the

Email addresses: karelmad@email.cz (Karel Maděra), smelko@d3s.mff.cuni.cz (Adam Šmelko), krulis@d3s.mff.cuni.cz (Martin Kruliš)

GPUs which is placed in contrast with the workload imbalance that arises from a straightforward parallel implementation of cross-correlation. Furthermore, the GPUs often suffer from data-throughput issues which are raised by the fact the GPU memory needs to feed tens of thousands of computing cores; hence, we need to design an algorithm that promotes sharing loaded inputs among the cores by cleverly caching data in shared memory or registers. Having a highly optimized, GPU-accelerated implementation of cross-correlation can be beneficial for many applications, especially when the inputs are large or when they need to be processed in real-time. Furthermore, the GPU can achieve a better watt-to-performance ratio than the CPU when used efficiently, so our effort can contribute to power consumption savings in the long run.

Each application of cross-correlation has slightly different parameters, depending on the size of the correlated signals or the number of instances being computed simultaneously. For instance, computing one instance of cross-correlation of two large signals would use a different optimization algorithm than computing a correlation between one small signal and a long sequence of medium-sized signals (i.e., searching for a pattern in a video sequence). Thus, our second aim is to compare and analyze the most typical applications of cross-correlation and find the best algorithm for each type.

There are basically two approaches to computing a cross-correlation. A naïve (or definition-based) implementation that directly follows the mathematical definition (with time complexity of $\mathcal{O}(N^2)$, where N is the size of both input signals) and an implementation that uses a Fourier Transform (FT) which has better asymptotical complexity ($\mathcal{O}(N \log N)$), but also higher computational overhead. We are focusing solely on the definition-based implementations where the actual code optimizations can be explored and which is more suitable for computing multiple instances of smaller signals. The FT-based algorithm can be implemented using highly optimized libraries like cuFFT, which is currently not interesting from the perspective of basic research in parallel computing and optimizations. However, we have implemented a cuFFT version of cross-correlation as well so we can compare and evaluate both approaches empirically.

1.1. Motivational application

Our research was motivated by material analysis — detecting material defects by electron microscope. The method uses the microscope to scan the surface of a material in a raster pattern. It projects an electron beam towards individual points in the raster and collects a *backscatter diffraction pattern* for each point. In computer science terms, the method collects a grey-scale image for each point in an input grid. For the selected material, there is a reference diffraction pattern that would be expected for a material without any defects. The collected images are compared with the reference image to detect possible distortions (e.g., translations, rotations, or warps) and these distortions can be interpreted as defects.

The comparison of measured and expected diffraction patterns is performed by dividing the images into multiple corresponding areas (i.e., areas with the same sizes and coordinates in both images) and cross-correlation is used to

determine a relative shift of these two areas in the images. Thus we need to compute a cross-correlation of N samples from one signal with N samples from M signals. The $N \cdot M$ easily reaches an order of millions, but the size of the areas is relatively small. Therefore, there is a lot of potential for parallelization, but it also means that the FT-based approach is likely to be slower than the definition-based approach.

Although the collection of the inputs from the microscope takes some time, the subsequent data processing can take even longer time when sequential implementation is used. Furthermore, in many cases, the results need to be recomputed multiple times with different input areas or different image normalization preprocessing. Therefore, a GPU-accelerated implementation would significantly improve the user experience when interpreting the data.

1.2. Contributions and outline

We have implemented, measured, and analyzed a wide range of optimizations of four of the most typical cross-correlation applications. The main contributions can be summarized into three points:

- We provide a CUDA-based algorithm (including an implementation) that is empirically evaluated as the best for each of the studied applications.
- Extensive evaluation and performance analysis of the individual optimization steps provide additional insight into GPU programming and code optimizations.
- We have determined the size thresholds of the input signals when the FT-based implementation (with better asymptotical complexity) takes over the definition-based implementation (which has lower overhead).

The source codes of the proposed algorithms, related scripts, and all the measured data as well as plotted graphs are available in a replication package in a GitHub repository [7].

The paper is organized as follows. The definition of cross-correlation including the formalization of the studied instances is presented in Section 2. Section 3 presents our analysis of parallelization possibilities and data re-use (inputs caching). The proposed algorithms and their implementation details are described in Section 4 and empirically evaluated in Section 5. Related work is overviewed in Section 6 and Section 7 concludes the paper.

2. Cross-correlation

First, we would like to review the mathematical definition of the cross-correlation (which is the basis for the definition-based implementation). Subsequently, we have selected and presented four of the most typical cross-correlation application types. Finally, we describe how the cross-correlation can be computed using Fourier transform.

2.1. Definition

Cross-correlation, also known as sliding dot product or sliding inner product, is a function describing the similarity of two series or two functions based on their relative displacement Cross-correlation of functions $f, g : \mathbb{C} \to \mathbb{R}$, denoted as $f \star g$, is defined by the following formula:

$$(f \star g)(\tau) = \int_{-\infty}^{\infty} \overline{f(t)} g(t + \tau)\, dt,$$

where $\overline{f(t)}$ denotes the complex conjugate of $f(t)$ and τ is the displacement of the two functions f and g. In simpler words, the value $(f \star g)(\tau)$ tells us how similar the function f is to g when g is shifted by τ, with a higher value representing higher similarity.

For two discrete functions, as will be used in our case, cross-correlation of functions $f, g : \mathbb{Z} \to \mathbb{R}$ is defined by the following formula:

$$(f \star g)[m] = \sum_{i=-\infty}^{\infty} \overline{f[i]} g[i + m],$$

This definition of cross-correlation can be extended for use in two dimensions, as is required, for example, in image processing. For two discrete functions $f, g : \mathbb{Z}^2 \to \mathbb{R}$, cross-correlation is defined as:

$$(f \star g)[m, n] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \overline{f[i, j]} g[i + m, j + n],$$

Even though cross-correlation is defined on the whole $\mathbb{Z}$ for one dimension and $\mathbb{Z}^2$ for two dimensions, most use cases of cross-correlation work only on finite inputs, such as image processing working on finite images. The only values we are interested in are those where the two images overlap, which restricts the computation to $(w_1 + w_2 - 1) \cdot (h_1 + h_2 - 1)$ resulting values, where w_i denotes the width and h_i denotes the height of the image i.

This limits the part of the output we are interested in and leads us to the time complexity of the *naïve* definition-based algorithm. For each of the $(w_1 + w_2 - 1) \cdot (h_1 + h_2 - 1)$ output values, we need to multiply the overlapping pixel values and sum up all the multiplication results. There will be at most $min(w_1, w_2) \cdot min(h_1, h_2)$ overlapping pixels. For simplicity, let us work with two images of the same size $w \cdot h$. Then the time complexity of the definition-based algorithm is $(2w - 1) \cdot (2h - 1) \cdot \mathcal{O}(w \cdot h)$, which gives us asymptotic complexity of $\mathcal{O}(w^2 \cdot h^2)$.

2.2. Forms of cross-correlation

In cross-correlation applications, several forms of computation can be found. Each enables different types of optimizations, such as data caching and data reuse, batching, or precomputing. These forms differ in the number of inputs and in the way cross-correlation is computed between the inputs. The four basic forms are depicted in Figure 1:

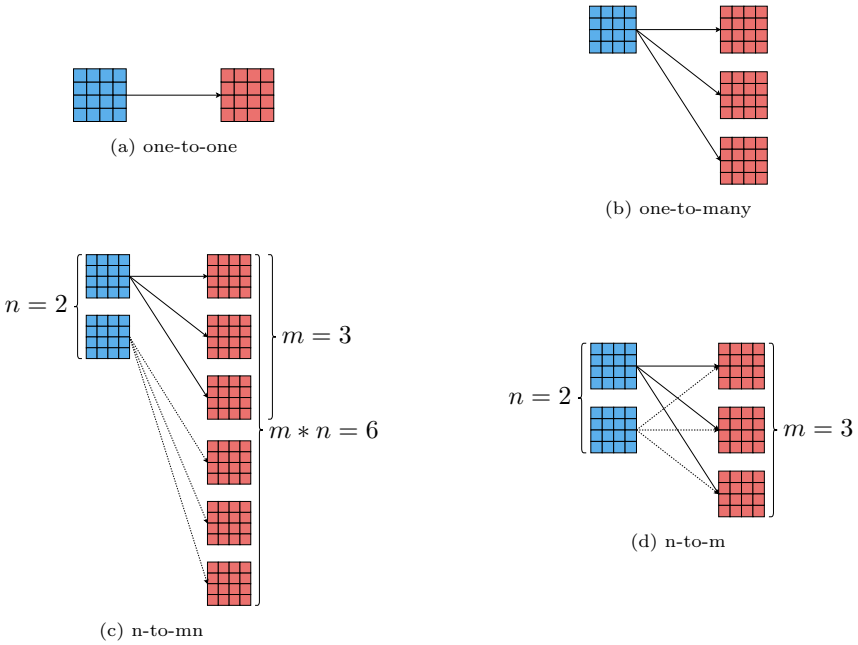


Figure 1: Basic forms of cross-correlation

1. one left input with one right input, in the rest of the paper referred to as *one-to-one* and depicted in Figure 1a;
2. one left input with many right inputs, referred to as *one-to-many* and depicted in Figure 1b;
3. n left inputs, **each one** cross-correlated with m **different** right inputs (multiple instances of *one-to-many*), referred to as *n-to-mn* and depicted in Figure 1c;
4. n left inputs, **all** cross-correlated with **all** m right inputs (full bipartite graph), referred to as *n-to-m* and depicted in Figure 1d.

The *one-to-many* form is typical for applications where one sample (a query) is located in a database or a time series of signal samples (like a video sequence). Similarly, *n-to-m* is merely an extension of this scenario where multiple queries are located in a database simultaneously [3]. Perhaps the most unusual pattern is *n-to-mn*. It has been inspired by the motivational application described in Section 1.1. It is an extension of *one-to-many*, where both the query and the database samples are divided into corresponding subsamples (e.g., areas with corresponding coordinates both correlated signals [6, 8]). We have observed even more complex forms in the applications; however, they did not present any more opportunities for parallel processing or caching optimizations.

While each pair of input matrices can always be computed independently, the *one-to-many*, *n-to-mn* and *n-to-m* types allow for the reuse of the left input matrix with multiple right input matrices, and the *n-to-m* makes it possible to reuse the right matrix for computation with multiple left input matrices.

For the same size of input data (x left and y right input matrices) the *n-to-m* requires the computation of $x \cdot y$ pairs of matrices, compared to the *n-to-mn* type which results in only y pairs. The increased level of parallelism and arithmetic intensity allows for additional optimizations of the *n-to-m* computation type compared to the *n-to-mn*. The *one-to-one* and *one-to-many* types are described separately, as compared to the general *n-to-mn* or *n-to-m* implementation, their implementations can more aggressively cache and reuse the left input matrix.

Implementations of the simpler types *one-to-one* and *one-to-many* can be extended to *n-to-m* or *n-to-mn* by running the simpler type of cross-correlation multiple times, possibly in parallel. Inversely, any implementation of either *n-to-m* or *n-to-mn* can be used to implement the two simpler types (with $n = 1$). Another type that we could consider is the computation of a large number of independent pairs, which can be implemented by *n-to-mn* (with $m = 1$). A large number of correlated pairs is a type not discussed further as it does not provide any additional opportunity for optimization compared to running the *one-to-one* several times in parallel.

In theory, more elaborate patterns of left-right matrix associations may be created. However, they can either be covered by a combination or iteration of the patterns described above and they provide no additional opportunities for data re-use that could be exploited by GPU hardware.

2.3. Computation using Fourier transform

There is an alternate algorithm for computing cross-correlation based on the discrete Fourier transform (DFT). The asymptotic complexity of this algorithm (in two dimensions) is $\mathcal{O}(w \cdot h \cdot \log_2(w \cdot h))$, where w is the width of each series and h the height of each series. This improves on the asymptotic complexity $\mathcal{O}(w^2 \cdot h^2)$ of the definition-based algorithm described in the previous section, but the actual complexity constants are higher (thus, the FT-based implementation is better only for inputs larger than a certain threshold).

The Discrete Fourier transform can only be used to compute a special type of cross-correlation, the so-called *circular* cross-correlation. For a finite series $N \in \mathbb{N}\{x\}_n = x_0, x_1, ..., x_{N-1}, \{y_n\} = y_0, y_1, ..., y_{N-1}$, circular cross-correlation is defined as:

$$(x \star_N y)_m = \sum_{i=0}^{N-1} \overline{x_m} y_{(m+i) \bmod N},$$

where $\overline{x_m}$ denotes the complex conjugate of x_m.

Based on the Cross-Correlation Theorem [9], the circular cross-correlation $(x \star_N y)_m$ can be computed using discrete Fourier transform (DFT) according to the following formula:

$$(x \star_N y)_m = \mathbb{F}^{-1}(\overline{\mathbb{F}(x)} * \mathbb{F}(y))$$

where $\mathbb{F}(x)$ and $\mathbb{F}(y)$ denote DFT of series x and y respectively, $\overline{\mathbb{F}(x)}$ denotes the complex conjugate of the DFT, $*$ denotes element-wise multiplication of two series and $\mathbb{F}^{-1}$ denotes inverse DFT.

To compute the non-circular (linear) cross-correlation of non-periodic series of size N, we pad both series with N zeros to the size $2N$, as indicated in Figure 2. The results of circular cross-correlation are then the results of linear cross-correlation, only circularly shifted by $N-1$ places to the left with one additional 0 value at index N.

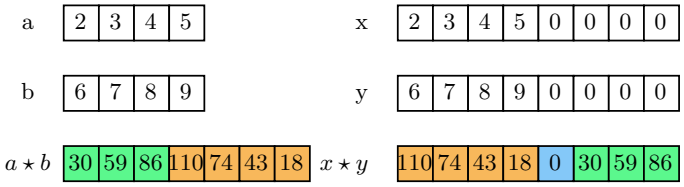


Figure 2: Comparison of linear and circular cross-correlation

This process can be expanded into two dimensions, where the matrices are padded with N rows and N columns of zeros before being passed through a 2D discrete Fourier transform. Here the circular shift of the results can be inverted by swapping the quadrants of the results while discarding row N and column N, which will be filled with zeros, as illustrated by Figure 3.

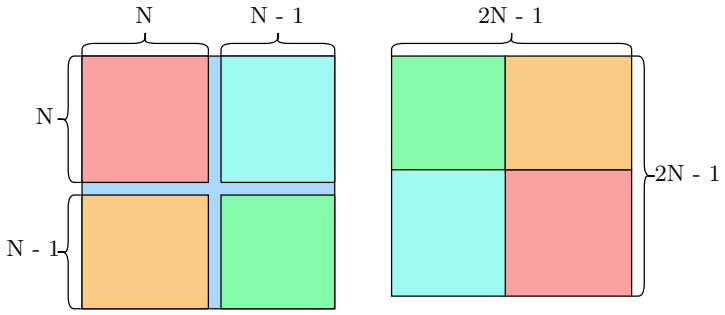


Figure 3: The result quadrant swap

Based on this description, we can deduce the time complexity of the algorithm. For two matrices $a, b \in \mathbb{R}^{h \times w}$, the steps of the algorithm are:

1. Padding $a_p, b_p \in \mathbb{R}^{2w \times 2h}$ of a and b with w columns and h rows of zeros in $\mathcal{O}(w \cdot h)$;
2. The Discrete Fourier Transform (DFT) $A, B \in \mathbb{C}^{2w \times 2h}$ of a_p and b_p in $\mathcal{O}(w \cdot h \cdot \log_2(w \cdot h))$;
3. Element-wise multiplication, also known as the Hadamard product, $C \in \mathbb{C}^{2w \times 2h} : C = \overline{A} \circ B$, where $\overline{A}$ the denotes complex conjugate of A, in $\mathcal{O}(w \cdot h)$;
4. Inverse DFT $c \in \mathbb{R}^{2w \times 2h}$ of C in $\mathcal{O}(w \cdot h \cdot \log_2(w \cdot h))$;
5. Quadrant swap in $\mathcal{O}(w \cdot h)$

In total, the steps described above give us an algorithm with asymptotic time complexity of $\mathcal{O}(w \cdot h \cdot \log_2(w \cdot h))$.

The FT-based algorithm will be used for comparison with the definition-based implementation. We have no ambition to optimize this algorithm further since the Fourier transform takes the most significant part and highly optimized libraries such as cuFFT[1] already exist.

3. Problem Analysis

The design and implementation of an optimal solution are affected by several aspects of the problem. Furthermore, different scenarios of computing multiple signals being cross-correlated simultaneously benefit from different approaches. In this section, we provide an overview of the most important optimizations which are essential in our proposed algorithms.

For the sake of simplicity, we will focus solely on 2D cross-correlation since 1D correlation is significantly less interesting and all the proposed optimizations can be extended into higher dimensions easily. The input signals are discrete, so we will refer to the materialized inputs as *matrices* to take advantage of the most familiar terminology available.

3.1. Workload parallelization

A single cross-correlation (one-to-one) produces one output matrix, where each element corresponds to one possible relative shift between the input matrices. An example with two 4×4 matrices is depicted in Figure 4. The value of the output element is computed as a sum of an element-wise multiplication performed on the overlapping area of the two input matrices.

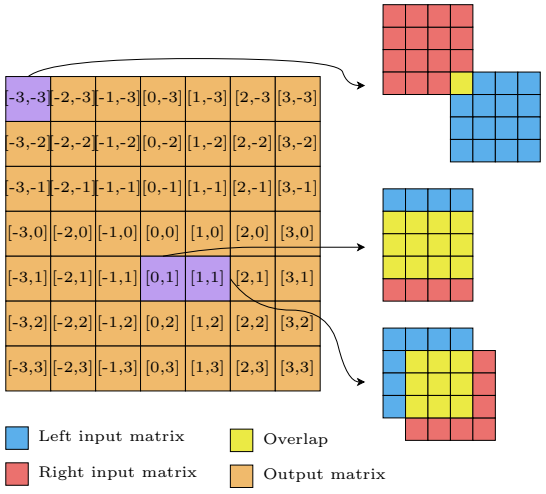


Figure 4: The output matrix with corresponding relative shifts of input matrices

[1] `https://docs.nvidia.com/cuda/cufft/`

The individual operations can be decomposed in a tree as indicated in Figure 5. The top-level node (green) represents the computation of a single output matrix. The second level (orange) represents computations of individual elements in the output matrix (different input overlaps). The third level (yellow) corresponds to the elementary multiplications performed on the overlapping area.

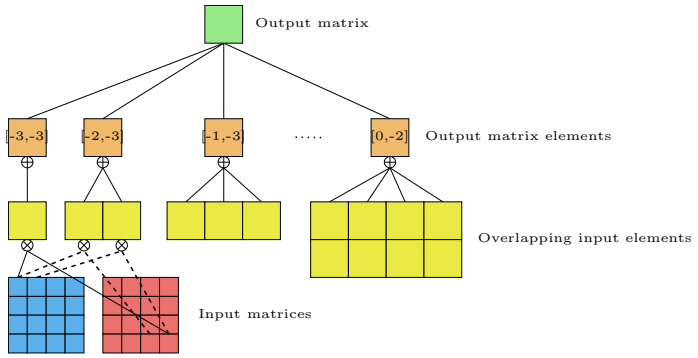


Figure 5: A work decomposition of one-to-one cross-correlation

Both the first and the second levels comprise independent operations — i.e., operations that can be performed without explicit data synchronization. The operations on the third level (multiplications) need to be reduced into the result at the second level (using a sum as the reduction operation). In case of more elaborate scenarios (*one-to-many*, *n-to-m*, and *n-to-mn*), the tree in Figure 5 is merely extended into a forest of independent trees, thus enabling another level of parallelism. Although this decomposition may indicate the problem is embarrassingly parallelizable, there are two major concerns for any GPU-accelerated implementation:

- The workload at the second level is highly irregular. Corner elements are computed by a single multiplication whilst the elements in the center of the output matrix require a full element-wise multiplication and sum-reduction of the input matrices. This imbalance may cause serious code divergence (i.e., suboptimal performance) in the warps and thread blocks.
- The problem is highly data-bound as each loaded element is used in a simple elementary operation (multiplication and addition). This might create a significant underutilization of the GPU cores if each core spends most of the time waiting for the data.

Therefore, our objective is to design algorithms that will heavily reuse loaded input data whilst attempting to provide a better workload balance, especially at the warp level.

3.1.1. Workload distribution

One of the key aspects that affect the efficiency of GPU-based parallel programs is workload decomposition and distribution among the allocated threads. It defines the level of parallelism (since the threads are executed concurrently), synchronization (when one thread needs to wait for the results of another thread), and native parallel cooperation (via shared memory or warp-level instructions). It also affects registry allocation and, transitively, data reuse (since the input data needs to be moved from global memory to registers).

To simplify the description of the subsequent optimizations and algorithms (especially their approach to data reuse), we adopt the *task* abstraction for the workload division and the work assignment to computing elements, where *task* is usually an element of work performed by one CUDA thread.

Task comprises a well-defined group of nodes from the work decomposition schema introduced in Figure 5. The most straightforward implementation would map one *overlap* (orange element at the second level) to one task and we denote this the **overlap-wise** (or simply the **overlap**) strategy. More elaborate task definitions, that would allow better input data reuse, will be outlined later using the overlap-wise strategy as the reference point.

In special cases, it is possible to assign each task to a group of threads (a warp or a block). This may provide a simpler approach to algorithms where more fine-grained division of tasks is impractical, but when each task may be processed in a cooperative or even SIMD-like manner. In the follow-up descriptions, we always assume that a task is assigned to one thread; unless we explicitly state otherwise.

3.2. One-to-one data reuse

The problem of the data-bound nature of the cross-correlation definition-based algorithm can be mitigated by smart caching of the input values which we generally refer to as *data reuse*. In other words, every input value loaded into the registers or shared memory should be reused in multiple computations (multiplications) to improve the ratio between load and arithmetic instructions. In the following, we will introduce several task-forming strategies (i.e., how the cross-correlation individual operations are assigned to CUDA threads) that are designed to enable data reuse.

The most straightforward strategy (**overlap-wise**) assigns one relative input overlap (one sum of overlapping products) to a CUDA thread. Analyzing the input data access patterns, we have made an observation, that adjacent overlaps share significant portions of input data (as illustrated in Figure 6).

The overlap-wise strategy can be implemented with data-reuse optimizations if the threads processing adjacent overlaps can share or exchange the input data efficiently (using shared memory or warp-shuffles). Another possibility is to create larger tasks that would aggregate multiple adjacent overlaps in a task so that a thread does not have to share the inputs with neighbors for data reuse as the reuse happens on loaded data internally. We have denoted this strategy **grouped-overlap**. For the sake of simplicity, we have selected only one

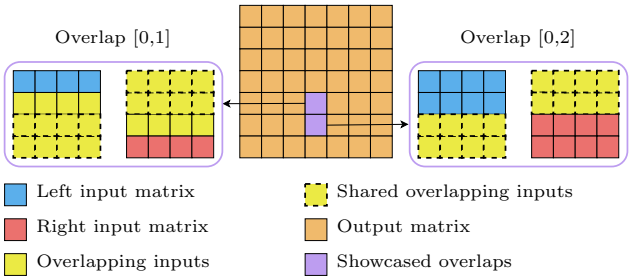


Figure 6: Input data shared between neighboring overlaps

dimension (which is depicted in Figure 6) where the task aggregates the overlaps with the same relative column displacement and adjacent row displacements. We have considered more complex aggregations as well as a col-wise approach, but using a row-wise approach to grouping is much simpler to implement and it promotes coalesced data loading[2] as we demonstrate later in Section 4.2.2.

3.3. Fine grained parallelism

In most cases (especially when multiple cross-correlations are computed simultaneously), the overlap-wise strategy gives us a sufficient amount of tasks to easily saturate the GPU. In fact, most of the data reuse strategies exploit some form of grouping — i.e., one task groups computations of multiple overlaps together. In the case of smaller instances (especially in one-to-one cross-correlation), the total number of tasks may decrease so that the GPU is no longer entirely saturated. In such cases, we can choose a more fine-grained workload division. Each overlap computation is basically an element-wise sum of a Hadamard product of the overlapping part of the input matrices (as indicated by Figure 5). The sum itself is associative, so we can compute the Hadamard product by multiple threads (concurrently) and then use a parallel sum reduction to get the result.

There are many ways to divide the Hadamard product (i.e., the matrix representing individual multiplications), perhaps the most straightforward is to divide the matrix of products into stripes of adjacent rows (of a constant height, except for the last stripe which may have fewer rows). This strategy is denoted **split-row** and the height of the row stripes can be selected as a tuning parameter of the algorithm. Analogically, we define **split-col** strategy, which uses the same approach but creates stripes of columns instead of rows. The selection of row or column orientation for striping depends mainly on the data layout, in traditional row-major layout, the **split-row** is better. In general, we refer to the principle of striping columns of rows as **split-overlap** (in case the distinction of orientation is not necessary or layout-dependent).

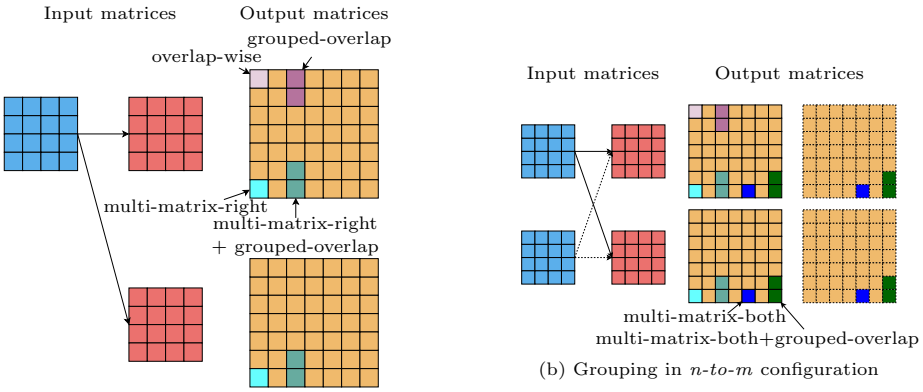
[2]Assuming the input matrices are stored in traditional row-wise layout.

Technically, the split-overlap principle can be combined with the previously mentioned data reuse grouped-overlap; however, this often turns out to be counterproductive. The split-overlap is used in case the GPU is not saturated and in such cases, creating enough work for the threads is far more important than data reuse (i.e., the subsequent grouping of the overlap stripes does not improve the performance).

An alternative approach to split-overlap is to use basic overlap-wise task definition but assign a whole warp of threads per task. The warp divides the individual task elements (i.e., the products) among the threads evenly, while each thread accumulates its partial sum in its register. Finally, the threads employ warp instructions for the final reduction. This approach is used in a specialized **warp-per-overlap** algorithm (Section 4.4).

3.4. Processing multiple inputs simultaneously

When multiple cross-correlations are being computed simultaneously (i.e., in *one-to-many*, *n-to-m*, and *n-to-mn* scenarios), another level of data reuse becomes available. We can create tasks that aggregate computations using the same overlaps (relative dislocations) but on different input matrices. The main advantage is that the sizes of the overlapping inputs are exactly the same so this strategy does not negatively affect load balancing. Based on the application scenario, we have decided to explore two possible strategies:



(a) Grouping in *one-to-many* configuration

(b) Grouping in *n-to-m* configuration

Figure 7: Multi-matrix approach and its combination with overlap grouping

- **multi-matrix-right** uses multiple right matrices for each overlap (i.e., each value loaded from the left matrix is used in multiple correlations with right matrices). This approach will work in all three multi-matrix scenarios (*one-to-many*, *n-to-m*, and *n-to-mn*).
- **multi-matrix-both** uses multiple left and right matrices so that each value from the left is used with all right matrices and vice versa. This approach is designed solely for the *n-to-m* scenario.

Figure 7 shows the multi-matrix strategies and compares them with the grouped-overlap. Furthermore, the multi-matrix strategies can be combined with the grouped-overlap strategy as well as the split-overlap strategy.

4. Proposed Solutions

We have experimented with various approaches to the problem and we present the best solution for each scenario. Most of the proposed solutions are based on a *warp-shuffle algorithm* which was designed to embrace smart data caching in registers and their exchange among neighboring threads using warp-shuffle instructions. The algorithm can be improved by several data reuse and work distribution optimizations described in the previous section. The warp-shuffle algorithm is not very suitable for very small inputs, so we also present a *warp-per-overlap* algorithm that prioritizes better workload distribution over data reuse which is critical when the GPU is not saturated by the warp-shuffle algorithm. The complete source codes with additional technical details are available in our replication package [7].

4.1. Warp-shuffle algorithm

The key principle of the warp-shuffle algorithm is that the threads within a warp[3] reuse data loaded from global memory into registers and employ warp-shuffle instructions to distribute the actual values. The same idea is behind the grouped-overlap reuse (depicted in Figure 6), but in this case, the data reuse is col-wise rather than row-wise and it takes place in the registers of the entire warp (not the registers of a single thread) which need to be updated by warp-wise instructions (warp-shuffles).

First, we demonstrate the main principle using an overlap-wise strategy where the jobs are distributed among the treads so that each warp processes consecutive overlaps on the same row in the output matrix. In other words, the threads of a warp will process overlaps $[x, y], [x + 1, y], \ldots, [x + 31, y]$ (where x is divisible by 32). Figure 8 illustrates which data (from the left and the right matrix) are used by two adjacent threads from a warp if the overlapped area is traversed in row-major order.

A quick analysis reveals, that the data from both matrices can be shared among the threads. In the case of the left matrix, the value used by thread 1 is required by thread 0 in the subsequent iteration. In general, thread i can load the value for the next iteration from thread $i + 1$, so technically, the data are being shifted to the left among the threads. In the case of the right matrix, each thread requires the same value in the same iteration, except for the corner cases where a thread gets out of the bound of its overlapping area.

[3] A group of 32 consecutive threads executed in lock-step.

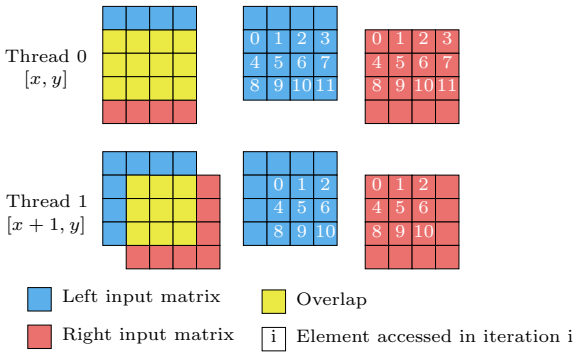


Figure 8: Input data shared between neighboring overlaps

4.1.1. Warp-wise buffers

Data from both input matrices are cached in warp-wise buffers. A *warp-wise buffer* is distributed among the registers of the individual threads in a round-robin manner and managed by warp-shuffle instructions. Each thread holds $N/32$ values, and value i is kept in the (i mod 32)-th thread of the warp.

The left matrix is buffered in a 64-item wide warp-wise buffer (2 registers per thread). Each thread t finds its current value on the index t, which is coincidentally also stored in the register of thread t. After each step, the buffer is shifted to the left by one item using (two) warp-shuffle instructions. The reason for using the 64-item wide buffer is to promote coalesced loading from the main memory (the data can be loaded in 32-item wide transactions instead of one by one). With 64 items, the buffer can be rotated 32× without accessing main memory and after that, the second half of the buffer can be replenished with a single coalesced load.

The right matrix is buffered in a warp-wise buffer of 32 items (one register per thread). This buffer does not require rotations, but the value required for each step needs to be loaded from the corresponding thread. This operation is also handled by a warp-shuffle instruction which can broadcast one value from the selected thread (i.e., its register) to all threads in a warp.

Figure 9 depicts data movements in 32 consecutive steps. Note that each step comprises one arithmetic (FMA[4]) operation and three warp-shuffle instructions. At the end of the 32-step block, both the gray part of the left buffer and the entire right buffer are filled with new data from the global memory in two coalesced transactions (when the entire warp loads a compact block of memory).

4.1.2. Technical details

There are several technical issues worth mentioning, albeit they are not essential for understanding the algorithm. The first one is how to handle corner

[4]Fused Multiply-Add — i.e., an instruction computing $x * y + z$ expression.

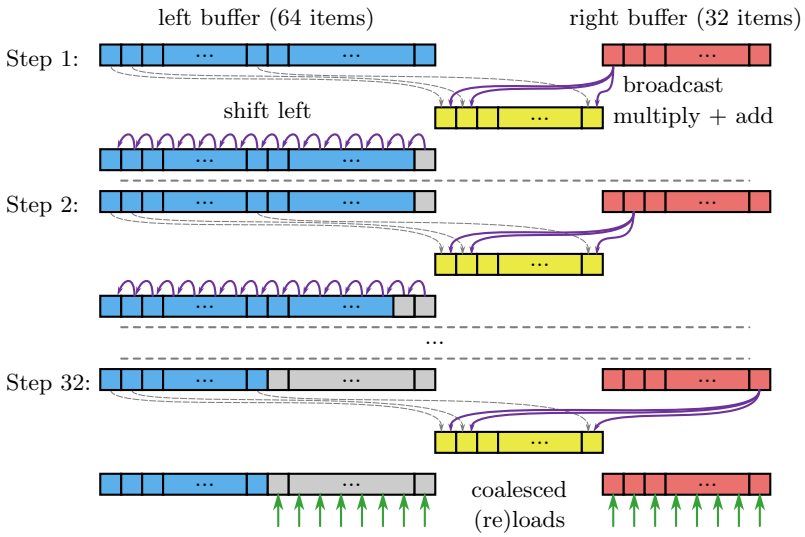


Figure 9: Buffering and data movements in warp-shuffle algorithm

cases, since the processed matrices are rarely aligned to the warp size and even if they are, the individual overlaps have different sizes. The loading of the inputs is handled in a way that values outside the input matrices are replaced with zeroes in the warp-wise buffers. Zeroes will not affect the cross-correlation results when used as inputs, so the only conditional code is in the loading step.

In the algorithm description, we have made a requirement, that consecutive items of the output matrix (on a row) are processed by a warp. In more detail, the thread block allocation is made so that the x-dimension has always size 32 (represents threads within a warp), and the number of warps (the y-dimension) in a block is a tunable parameter of the algorithm (allowing us to find the best occupancy of the multiprocessors). Subsequent warps in a block operate on subsequent rows, which slightly improves the hit rate in hardware caches and also becomes beneficial in the follow-up optimizations. Furthermore, aligning the workload to the warp size also ensures that the edge case (when part of the warp is outside the output matrix) does not create any additional problems in the input buffering mechanism.

Finally, we made some special steps to ensure that the most internal loop of the algorithm (which performs exactly 32 steps) is unrolled, so it can be highly optimized by the compiler (e.g., by resolving constant expressions in indices). The required hacks can be found in our source code.

4.2. One-to-one optimizations

The warp-shuffle algorithm can be improved by data reuse techniques described in Section 3. We shall start with optimizations that do not require multiple inputs (i.e., addressing *one-to-one* configuration).

4.2.1. Split-overlap

The split-overlap (specifically the split-row) optimization is designed for situations where the number of tasks is not sufficient to saturate the GPU and we need to decompose the workload further to enable another level of parallelism. The warp-shuffle algorithm is designed to be fully compatible with this optimization. The only difference is that multiple warps are allocated for tasks, that would be processed by a single warp in a regular overlap-wise strategy. The Hadamard product of each result is then computed by multiple threads, each of them being assigned the same amount of rows of the overlapping area (except for the last thread which may receive less). The product is both associative and commutative, so we can compute its part concurrently. Given the number of individual fragments is intended to be relatively low, aggregation by `atomicAdd` operation is quite adequate in this case.

We have experimented further with better and more complex work distribution patterns that could reduce the time when individual threads are idle due to code divergence caused by the irregularity of the workload. However, the overall improvement was barely measurable, possibly due to the fact the improvements were partially outweighed by increased overhead computations (calculating indices). Thus, we have concluded more elaborate solutions are not worth pursuing further, especially given the increase in their coding complexity.

4.2.2. Grouped-overlap

If the inputs are sufficiently large, we can take an opposite path to optimizations that decrease the number of tasks due to grouping but promote data reusability further. The warp-shuffle approach already reduces global memory loads whilst making them more coalesced, but the data still needs to be shuffled among the threads. The profiling of basic overlap-wise implementation of the warp-shuffle algorithm confirms what is also apparent from Figure 9 — each FMA instruction (that performs the actual computation) requires 3 warp-shuffle instructions (that just ensure the data are in the right registers). We can improve this ratio by grouping overlaps (each task comprises multiple overlaps) as suggested in Section 3.2. It enables caching multiple rows from both left and right matrices and subsequently using each value loaded in registers in multiple FMA operations.

Figure 10 depicts a situation, where each task computes three overlaps displaced by one in the vertical direction ($[x, y]$, $[x, y + 1]$, and $[x, y + 2]$). The number of rows cached from the left matrix was also set to 3 (although we may choose a different number of rows than the number of grouped overlaps, the best performance is usually achieved when they are the same). The number of the right rows needs to be set as the number of grouped overlaps and the number of left rows minus one ($3 + 3 - 1 = 5$). This setup was selected so all rows cached on the left are loaded exactly once (rows on the right are loaded multiple times).

In this particular case, a thread performs 9 FMA instructions in each step while the registers are managed by 5 broadcasts and 6 shifts. In other words, the FMA to warp-shuffle instructions ratio was improved from $1 : 3$ to $9 : 11$.

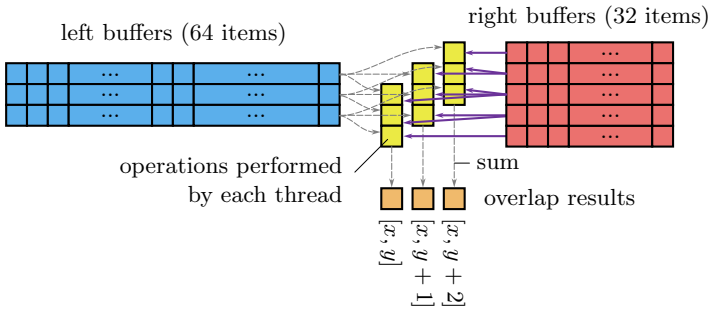


Figure 10: Data reuse in grouped-overlapped optimization (3 grouped overlaps)

By increasing the grouping (and caching) factor, we can achieve better ratios of FMA to shuffle instructions. On the other hand, grouping reduces the number of tasks and increases the required amount of registers per thread, which may lead to low occupation of GPU cores. The best factor was determined empirically as 4 overlaps per task.

Finally, there is one important implementation detail. Overlaps grouped together usually do not have the same size. As we iterate over the rows in the left matrix, the beginning or the end of the iteration needs to handle some corner cases. To reduce code divergence, we have divided the row-loop into three phases — the *init* phase, when the common overlapping part is growing, the *main* phase, which was described above, and the *final* phase, when the common overlapping part is diminishing. We have selected to implement the *init* and the *final* phases separately, to eliminate unnecessary conditions from the *main* phase code (which is usually the dominant part of the calculation).

4.3. Multiple cross-correlations

Another level of parallelism and data reuse is opened when multiple cross-correlations are computed simultaneously. The warp-shuffle principle remains intact, but each task aggregates the same work from multiple cross-correlations (duplicating the necessary input buffers and the output values). The greatest advantage is that the computations are truly independent and they have identical shapes, so no additional corner cases have to be handled. The only requirement is that the matrices being correlated simultaneously have the same dimensions.

4.3.1. Multi-matrix-right (one-to-many, n-to-mn)

The first type assumes that one left matrix is correlated with multiple right matrices. This holds for *one-to-many* and *n-to-mn* scenarios. Technically, this assumption holds also for *n-to-m* case, but we can do better optimizations with it as we demonstrate later.

Similarly to the grouped-overlap optimization, the objective is to improve the ratio between FMA instructions and warp-shuffle instructions. Increasing

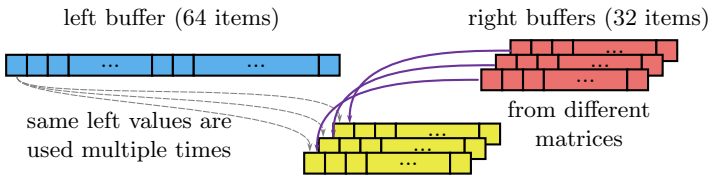


Figure 11: Data reuse of multiple right matrices

the number of right matrices by one adds one FMA instruction and one shuffle instruction as well, thus improving the ratio. In general, having r right matrices cached simultaneously, the ratio of the instructions will be $r : r + 2$ (i.e., approaching $1 : 1$ for large r values).

4.3.2. Multi-matrix-both (n-to-m)

In the case of *n-to-m* scenario, we correlate multiple left matrices with multiple right matrices, so we can employ caching and data reuse on both sides. Unlike the case of the *grouped-overlaps* optimization, all registers from the left buffer are multiplied with the broadcasted data from the right buffers which leads to $l \cdot r$ FMA operations per step (assuming l and r represents the amount of left and right cached matrices respectively). The details are depicted in Figure 12.

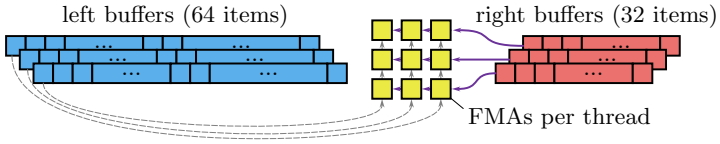


Figure 12: Data reuse when multiple matrices are used both on the left and the right

With this configuration, we can easily achieve a better ratio of FMA to shuffle instructions than $1 : 1$. For instance, in the case of 4 left and 4 right matrices being correlated in a joined effort, each thread will compute $4 \times 4 = 16$ FMA instructions for every $4 \times 2 + 4 = 12$ shuffle instructions. On the other hand, each intermediate result requires a separate register where the products are accumulated, which can easily create register pressure if higher values of l and r are selected.

4.3.3. Combining optimizations

One of the greatest advantages of *multi-matrix* extensions is that they are orthogonal to *split-row* and *grouped-overlap* optimizations — in other words, we can combine these two techniques to improve performance further. The combination with the *split-row* is completely straightforward and requires no special modifications. The combination with *grouped-overlap* is slightly more difficult to imagine, so we include a visual aid.

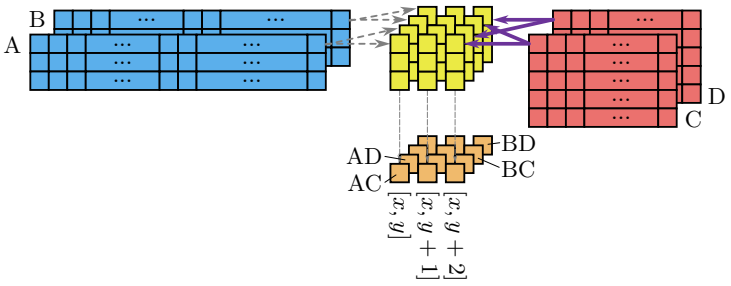


Figure 13: Combining multi-matrix-both with grouped-overlap

Figure 13 depicts the computation of a single thread when the two optimizations are combined. The input buffers are merely replicated in two dimensions — i.e., multiple rows from multiple matrices are being cached. The most difficult part is to find the right balance of the parameters to achieve optimal performance.

4.4. Warp-per-overlap algorithm

In the final part, we would like to revisit the problem of GPU underutilization when the matrices are very small. There is an alternate approach to the *warp-shuffle* algorithm with *split-row* optimization (described in Section 4.2.1), which may provide even better performance since it aims specifically to better core occupation and minimization of code-divergence.

One of the greatest benefits and limitations of the warp-shuffle algorithm is the requirement that all threads in a warp must process adjacent overlaps within one row. Although it enables coalesced load and data shuffles, it also becomes a source of great thread divergence when the matrices are rather small, especially when the overlapping area width is comparable with the warp size, but not exactly matching. For instance, when correlating two 17×17 matrices, the overlapping area is 33×33, Hence, the second warp on each row will compute only one result value. This limitation is not mitigated by the split-row optimization.

An alternate approach is to allocate an entire warp to process each task — a *warp-per-overlap* algorithm. Using the basic overlap-wise task division, a task work is to iteratively process all compute elements of the Hadamard product of the overlapping area. The iteration can be divided in a vectorization manner and the threads will be assigned the FMA operations using the round-robin principle (Figure 14). This way, the workload is more evenly distributed among the threads of a warp which minimizes code divergence and maximizes the effective core occupancy. On the other hand, this task allocation does not provide any means for data reuse and the non-regular data access pattern will require much more global memory transactions.

We have experimented with several memory optimizations that will make the data loads more coalesced as well as manual caching of the input data in

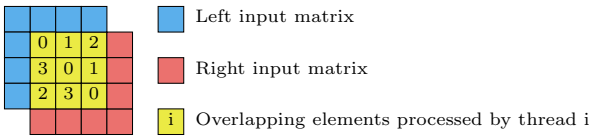


Figure 14: The warp-per-overlap principle demonstrated on warps of size 4

the shared memory. Although these improvements may be theoretically intriguing, they are not practical. The reason is that additional modifications of this algorithm increase the coding complexity quite a bit whilst improving the performance only when the input matrices are quite large. However, for larger input matrices the previously presented warp-shuffle algorithm exhibits better performance. Therefore, we propose the warp-per-overlap algorithm as an alternative to the split-row algorithm only for very small input matrices.

5. Experimental Results

To evaluate the performance of the proposed algorithms and optimizations, we performed extensive benchmarks covering the vast configuration space of optimization combinations, input sizes, and individual attuning parameters. In this section, we summarize the results of the described optimizations and compare them with each other to find the optimal one for each input size and problem configuration. We also compare our methods with the asymptotically better FFT-based algorithm to find the limits of definition-based algorithms.

5.1. Experimental setup

We carried out the experiments on three different NVIDIA GPUs representing three different architectures: Tesla V100 SXM2 32 GB (Volta arch.), Tesla A100 PCIe 80 GB (Ampere arch.), and Tesla L40 PCIe 48 GB (Ada Lovelace arch.). The systems were using CUDA 12.2 with driver version 535.104.05. Each single benchmark was repeated 10 times, each time running for at least 0.1 seconds. This setup was necessary to ensure proper measurement of short-running kernels. After removing the outliers using the IQR method[5], we have performed basic statistics computing the arithmetic mean and the standard deviation (SD) for each experiment individually. The means are presented as the results, the average ratio of mean and SD over the experiments is 0.43%, which indicates good stability of the measurements so we decided not to include SD values in graphs. All benchmarking datasets were synthetic, with data sampled randomly from the same uniform distribution. The performance of the benchmarked algorithms is not data-dependent.

The relative results (speedups of individual optimizations) do not differ significantly among the three GPU architectures. The figures further presented in

[5]Considering IQR being the range between Q_1 and Q_3 quantiles ($IQR = Q_3 - Q_1$), outliers are points below $Q_1 - 1.5 \cdot IQR$ and above $Q_3 + 1.5 \cdot IQR$.

the paper cover only the results of the Tesla A100 GPU. The complete result set is available in our replication package [7].

5.1.1. FFT-based algorithm

In the following discussion, we also compare our proposed algorithms with the FFT-based approach. As described in Section 2.3, the FFT-based algorithm runs in 5 steps: Input padding, Discrete Fourier Transform (DFT), Hadamard product, Inverse DFT and quadrant swap. We used highly optimized cuFFT routines for DFT and Inverse DFT. The Hadamard product was implemented in a custom kernel since it is an embarrassingly parallelizable algorithm. We chose to omit the quadrant swap from the measurements since this step is not generally required for all cross-correlation use cases (e.g. when the correlation result is processed further and the swap can be amortized in a subsequent step).

The DFT routines operate on a *cuFFT plan* — an opaque data structure, which needs to be initialized beforehand. Although we can only speculate what operations the plan initialization exactly performs, cuFFT documentation states that it also allocates GPU memory. The allocation is quite time-consuming in comparison to the kernel execution, so we decided to plot it separately to provide a more accurate picture of the relative performance. We plotted the FFT-based algorithm performance in two variants — *fft* aggregates the runtime of DFT, Hadamard product, and Inverse DFT; the *fft+plan* also includes the time required for the plan creation. The *fft+plan* presents a time that would be closer to a realistic application. The *fft* shows the theoretical limits of the FFT-based approach that may be relevant if the initialization phase can be amortized or the cuFFT plan data structure can be reused for many computations.

5.2. One-to-one benchmarks

The one-to-one configuration can be solved using either the *warp-shuffle* algorithm (possibly with the *grouped-overlap* or *split-row* optimization) or the *warp-per-overlap* algorithm. First, we evaluate *grouped-overlap* and *split-row* optimizations separately to determine optimal tuning parameters. Afterward, we present the overall comparison of all methods including the naïve *overlap-wise* algorithm (baseline) and the FFT-based algorithm.

5.2.1. Grouped-overlap

In this micro-benchmark, we present normalized times per single FMA operation (with amortized data transfers). To ensure reasonably interpretable values, we needed to saturate the GPU cores completely (i.e., generate enough workload even if the inputs are small). This prevents a misleading observation when an increase in the grouping factor (which improves efficiency) could be perceived as a decrease in the apparent FMA throughput just because the GPU gets undersaturated. We manually modified the kernel of the algorithm to run 4000 copies of the input problem in a single grid. This allows us to correctly measure the actual FMA throughput, but the results are not directly comparable with other methods, especially in the case of very small inputs.

The results are shown in Figure 15. By increasing the number of grouped overlaps (and hence increasing the caching and data reuse factor), the optimization performs better for all matrix sizes. We can also observe that the algorithm performs significantly better for larger inputs, which is caused by the inherent limitation of the warp-shuffle principle. Very small matrices (like 16×16) struggle with GPU underutilization and code divergence as the threads in a warp become idle for a considerable amount of time. More specifically, for the 16×16 matrix, cumulative idle thread time (i.e., the amount of cycles a thread does not contribute to the computation) is on average 50% for all warps. With the larger inputs, the thread utilization becomes better and once exceeding 64×64, the warp-shuffle algorithm can fully utilize the warp-wise buffer for the left matrix (as described in Section 4.1). The idle thread time per warp averages to 16% for 128×128 matrix, asymptotically nearing 0% as the size increases.

For the sake of brevity, we do not show the plot measuring the influence of the block size on the performance of the algorithm (it is available in our replication package). The results conclusively show that as soon as the block size reaches a sufficient level to fully utilize a GPU streaming multiprocessor, the performance of the algorithm is not affected by increasing it further. Therefore, we used the block size of 4 warps (4×32) for all the experiments.

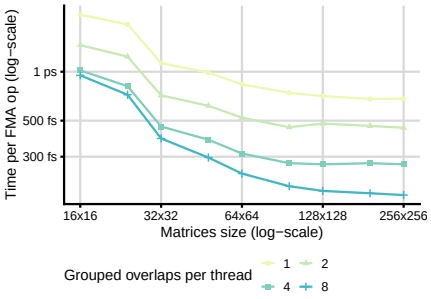


Figure 15: The *grouped-overlap* results, normalized times (per FMA) for completely saturated GPU

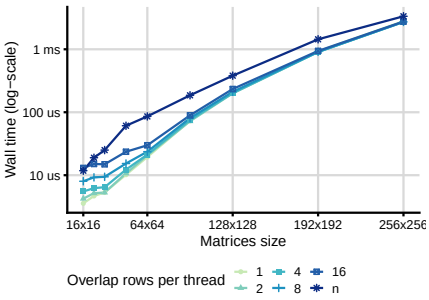


Figure 16: The *split-row* benchmark on various inputs, absolute (wall) times

5.2.2. Fine-grained parallelism

When problem size is not sufficient to saturate the GPU, a fine-grained parallelism is required. One possibility is to employ the *split-row* optimization for the warp-shuffle algorithm, which splits each Hadamard product into multiple independently processed stripes. Figure 16 shows the performance for different job granularity levels ranging from the finest job of 1 row per thread to n (all) per thread (no splitting takes place — i.e., referring to basic warp-shuffle implementation). As expected, the finest granularity helps the most for the smallest matrices and the speedup over n (baseline) variant progressively diminishes as the input size increases (and thus saturates the GPU without splitting).

The alternate approach (*warp-per-overlap* algorithm) has no tuning parameters, so we do not provide a separate micro-benchmark for it. The comparison of both algorithms is evaluated in the following.

5.2.3. Comparison of all one-to-one solutions

Figure 17 (left) summarizes the performance of the discussed one-to-one algorithms. The *baseline* algorithm denotes the naïve *overlap-wise* implementation (one thread per one overlap with no data reuse) which we use as a baseline. Algorithms, which have tuning parameters, use their optimal values for given input sizes (as determined in the previous micro-benchmarks).

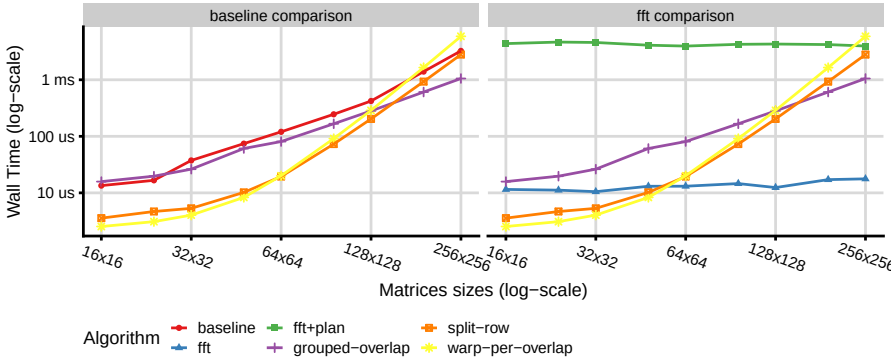


Figure 17: Comparison of one-to-one algorithms

The *grouped-overlap* optimization is the most beneficial for larger matrices while for smaller matrices it suffers the low GPU occupancy due to the insufficient amount of tasks. The *split-row* and *warp-per-overlap* algorithms perform better on smaller matrices as they resolve the occupancy issue. The *warp-per-overlap* performs better on very small inputs as it was designed specifically to prefer core occupancy over data caching. The *split-row* optimization of the *warp-shuffle* algorithm performs slightly worse for matrices smaller than 64×64; for larger matrices, the data reuse and coalesced loads become more important, so it outperforms *warp-per-overlap*. Overall, the proposed optimizations perform better than the baseline *overlap-wise* algorithm, being 5.3× faster for 16×16 input and 3.1× faster for 256×256 input.

The right part of Figure 17 reveals that the cuFFT plan creation is the most costly part of the algorithm, dominating the runtime in each measured data point. When the initialization is taken into account, the definition-based approach appears much better in terms of performance. The turning point, where the *fft+plan* surpasses our optimizations, seems to be around 384×384 matrix. When considering *fft* alone, only the *warp-per-overlap* algorithm outperforms it (having 4.5× speedup on 16×16 inputs) and the turning point is around 48×48.

5.3. One-to-many benchmarks

The *one-to-many* and *n-to-mn* scenarios enable utilization of the *multi-matrix-right* optimization of the warp shuffle algorithm. This optimization can be combined with *grouped-overlap* or *split-row*, so we present their respective performance evaluation in detail.

We did not include the *warp-per-overlap* evaluation in this section, because it does not provide any additional improvement in terms of performance. The additional workload of multiple cross-correlations mitigates the need for extremely fine-grained parallelism, so the *split-row* optimization is more than sufficient even for the smallest matrices.

5.3.1. Multi-matrix-right with grouped-overlap

In this configuration, we are benchmarking the one-to-many scenario with 4000 right matrices, which completely saturates the GPU. The left subplot of Figure 18 shows the *grouped-overlap* results without the *multi-matrix-right* optimization. In the middle and the right subplot, the number of right matrices per thread is 2 and 4 respectively (i.e., enabling the multi-matrix caching). The results indicate that increasing the number of right matrices per thread does not collide with the data reuse made by the *grouped-overlap* optimization and both optimizations can work in synergy.

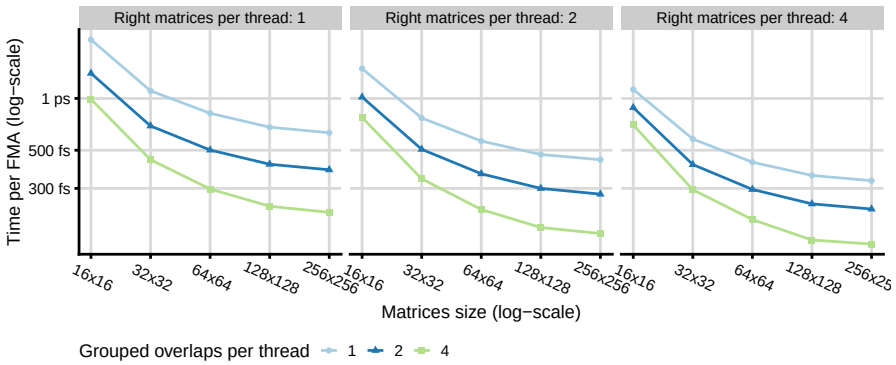


Figure 18: *Multi-matrix-right+grouped-overlap* results (one-to-many, 4000 right matrices)

Considering a sufficient total number of the right matrices, we can increase the factor of right matrices per thread significantly more and still expect the performance to improve. The primary limitation is the maximum number of registers per thread a GPU allows to allocate. The required number of registers increases linearly with the product of right matrices per thread used by *multi-matrix-right* and warp-wise buffers used by the *grouped-overlap* (which is about $3 \cdot 4^2$ registers per thread for the variant that reuses the data the most intensively in Figure 18). When the maximum is exceeded, the GPU resorts to register spilling (offloading to local memory), which harms the performance significantly.

We have observed that the parameter values presented in Figure 18 are in a reasonable range. Increasing the grouping factor or number of right matrices further does not help much with performance on current GPU architectures, but it creates additional issues with the compilation (especially bloating the size of our artifact). Hence, we have excluded higher values from the presented results for practical reasons.

5.3.2. Multi-matrix-right with split-row

This micro-benchmark was designed to determine how the combination of multi-matrix data reuse and fine-grained parallelism can improve performance. In theory, applying *multi-matrix-right* on small inputs may decrease the performance because it groups tasks, thus limiting the parallelism. Combining multi-matrix optimization with *split-row* may provide enough parallel GPU work whilst improving the data reuse.

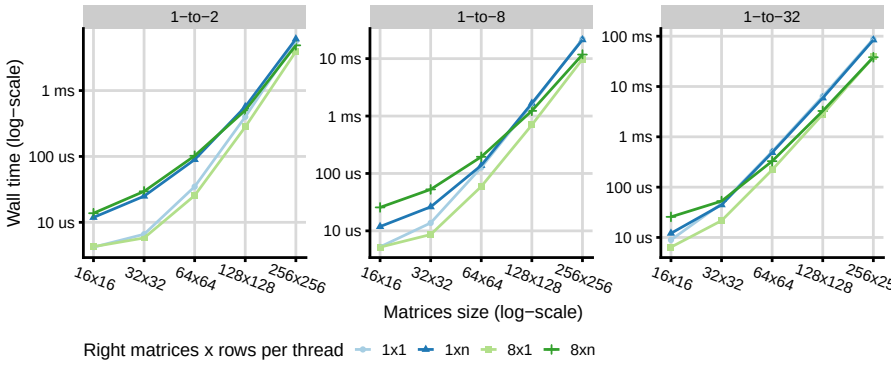


Figure 19: *Multi-matrix-right+split-row* benchmark results (please note that the 8×1 and $8 \times n$ parametrizations are in fact 2×1 and $2 \times n$ in the *1-to-2* scenario since we can cache only up to the total number of right matrices)

Figure 19 demonstrates how the *split-row* improves performance for small problem sizes. The $1 \times n$ and $8 \times n$ denote the versions that do not take advantage of *split-row* (the size of row-stripes is n, which stands for the size of the overlapping area). The 1×1 and 8×1 stand for the most fine-grained versions of *split-row* (one task takes only one row). The data indicate that in the extreme, the speedup caused by splitting the rows could reach an order of magnitude (16×16 with a low number of right matrices). Furthermore, the 8×1 parametrization (i.e., the most fine-grained division that caches 8 right matrices) exhibits the best performance over the examined domain.

5.3.3. Comparison of one-to-many optimizations

The overall comparison is presented in Figure 20. Similarly, as for one-to-one optimizations, the *split-row* dominates the small matrices and *grouped-overlap* dominates the larger matrices. Employing *multi-matrix-right* (especially when

combined with *split-row*) shifts the turning point where higher data reuse wins over more granular jobs. Using 32 right matrices, we achieve 11.8× speedup over naïve *overlap-wise* (baseline) algorithm for 16×16 input and 6× speedup for 256×256 input.

When we compare the best definition-based algorithm with cuFFT, the *split-row* still outperforms *fft* for extra small matrices. The turning point for *fft+plan* is slightly beyond the size of 256×256 for 2 input matrices, and 128×128 for 32 input matrices.

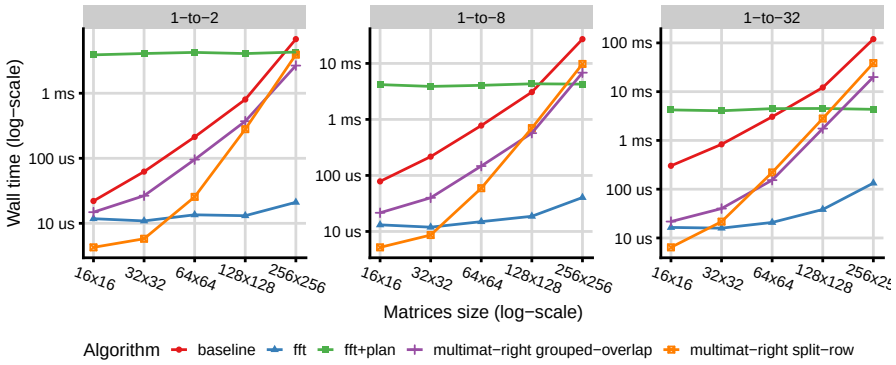


Figure 20: Comparison of one-to-many algorithms

5.3.4. Extending one-to-many into n-to-mn

The *n-to-mn* problem is in fact n instances of *one-to-many* problem. There are two ways of extending the *one-to-many* implementation — we could either simply run the original kernel n times simultaneously or create a new kernel that takes an additional index. After a careful analysis, we found no additional benefits of implementing a separate kernel. When running *one-to-many* kernel n times, the only issue worth mentioning is that the runtime must utilize a sufficient amount of CUDA streams, so the execution of the kernels may overlap in case the individual invocations cannot saturate the GPU.

The overhead of the simultaneous kernel execution is negligible, so we have omitted figures with the performance results from the paper for the sake of brevity. The data and the plots may be found in the attached replication package.

5.4. n-to-m benchmarks

This scenario allows the most elaborate data reuse pattern called the *multi-matrix-both* optimization. Similarly to *multi-matrix-right*, it can be combined with *grouped-overlap* or *split-row*.

5.4.1. Multi-matrix-both with grouped-overlap

In Figure 21, we present the results of *grouped-overlap* alone (left subfigure), combined with *multi-matrix-right* (center subfigure), and with *multi-matrix-both* (right subfigure). Regardless of the number of grouped overlaps, the *multi-matrix* optimization alone improves the speedup, and the combination of both optimizations exhibits the best performance. In the case of the highest overlap grouping, the speedup of *both* variant over *right* variant is about 1.75× on all input sizes.

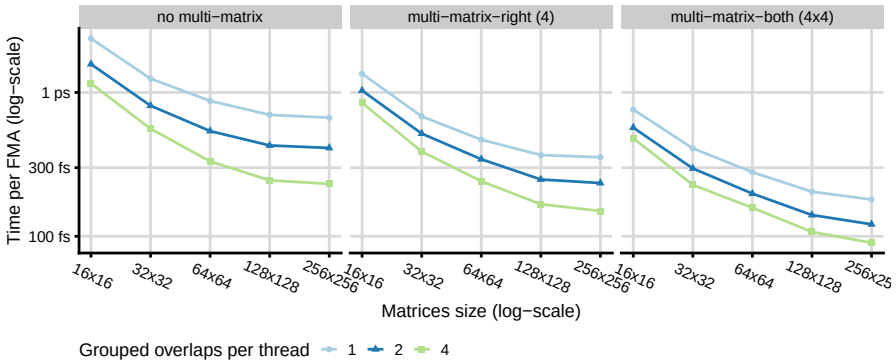


Figure 21: *Multi-matrix-both+grouped-overlap* benchmark results (128-to-128 matrices)

5.4.2. Multi-matrix-both with split-row

Similarly to *multi-matrix-right* combination, we aim at verifying that *split-row* optimization enables the data reuse on smaller matrices without any performance downgrade. We tested this on two different matrix counts: 2-to-2 and 8-to-8 matrices (top and bottom pair of subfigures in Figure 22 respectively). The results indeed show that for small matrices, the *multi-matrix* alone (the left pair of subfigures) is slower than the *multi-matrix* combined with *split-row* (the right pair of subfigures). The speedup of finer parallelism for 16×16 matrix and the highest — about 5×. As expected, the speedup gets negligible for larger matrices.

5.4.3. Comparison of n-to-m optimizations

The overall comparison is presented in Figure 23. Similarly to *one-to-many* optimizations, the *split-row* dominates smaller inputs while *grouped-overlap* dominates larger inputs. However, when the multi-matrix factor gets higher (32-to-32 matrices), the *grouped-overlap* gets more efficient than *split-row* as the GPU is already saturated and data reuse becomes more important.

Another observation is that cuFFT gets better even for slightly smaller matrices when the number of cross-correlations is growing. That is a natural conclusion of the fact that the cuFFT plan initialization takes constant time, so it

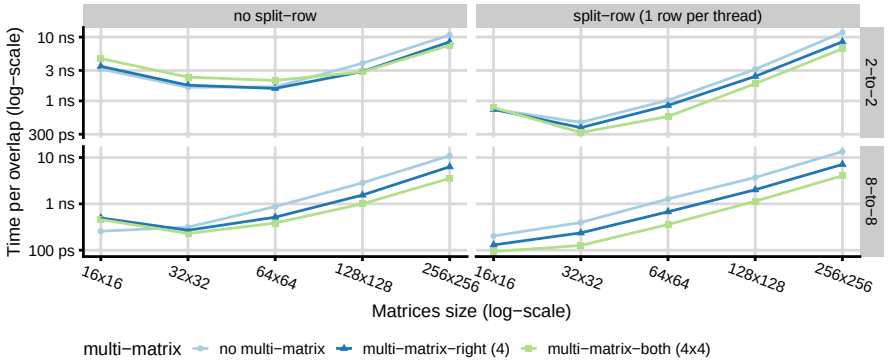


Figure 22: *Multi-matrix-both+split-row* benchmark results

gets more amortized into the overall computation. For 32-to-32 matrices, the turning point gets as low as 64×64 matrices.

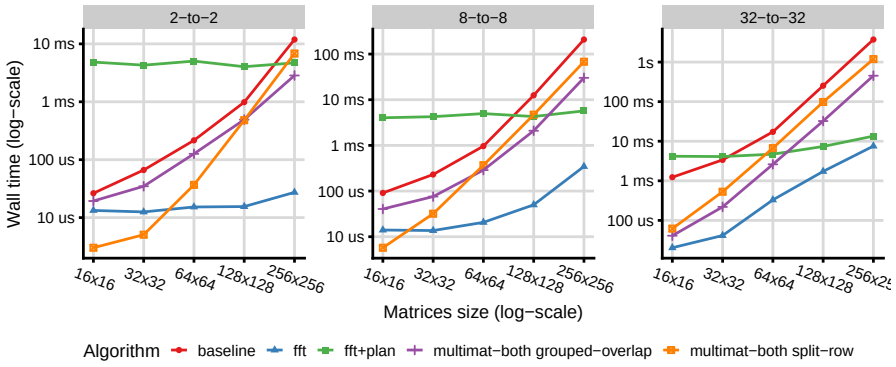


Figure 23: Comparison of n-to-m algorithms on various inputs

5.5. Summary and outcomes

To summarize the empirical results, we provide basic guidelines for selecting the optimal algorithm and its optimization. Table 1 presents the algorithm of choice for given scenarios (rows) and matrix sizes (columns). The *one-to-many* instances implicitly assume utilization of *multi-matrix-right* optimization and the *n-to-m* always employ *multi-matrix-both* optimization.

As indicated in the previous benchmarks, the smallest configurations benefit from *split-row* optimization (or the *warp-per-overlap* algorithm, in case of *one-to-one* scenario). The middle-sized problems can benefit from the *grouped-overlap* optimization and the largest problems should switch to the FFT-based approach which is asymptotically better.

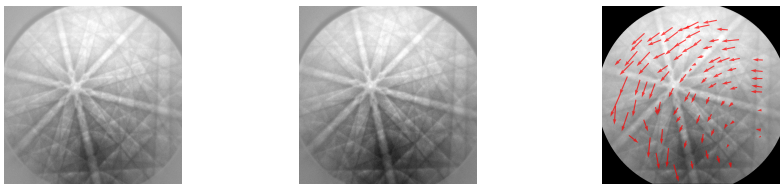
	16^2	32^2	64^2	128^2	192^2	256^2	384^2	∞
1-to-1	warp-overlap			split	grouped			fft+p
1-to-2	split				grouped		fft+p	
1-to-8	split			grouped			fft+p	
1-to-32	split		grouped			fft+p		
2-to-2	split				grouped		fft+p	
8-to-8	split			grouped			fft+p	
32-to-32	grouped				fft+p			

Table 1: The best algorithms (and optimizations) for individual scenarios and input sizes

Please note that the turning points for each configuration are not exact and they may differ slightly across the GPU architectures.

5.6. Application on real-world data

To verify the synthetic datasets are sufficient for performance measurements, an additional experiment on a real-world dataset was performed. We used a problem of material deformation from electron microscopy (described in Section 1.1). Figure 24 shows a use case of finding a deformation pattern in a FeAl alloy. The reference pattern (Figure 24a) is cross-correlated with the deformed pattern (Figure 24b), and the output is post-processed to be visualized as a displacement in various regions (Figure 24c). Our testing input consisted of a single reference pattern and 50 deformed patterns, each divided into 86 subregions of size 96×96 (i.e., a *n-to-mn* configuration where 86 tiles are correlated with 50×86 tiles).



(a) Reference pattern (b) Deformed pattern (c) Deformation visualization

Figure 24: A visualization of FeAl alloy deformation computed using cross-correlation

The runtime of the computation took about 79 milliseconds on A100 using multi-matrix-right and grouped-overlaps optimizations, which matched the observation when synthetic data was used in the same input configuration. Also, we measured the error of our approach using single-precision arithmetic compared to the original Python script used to compute material deformation (which uses double-precision). The mean relative difference between the elements of cross-correlation output was 2.39×10^{-6} with the maximum point difference of 3.8% and the variance of 8.49×10^{-6}. This shows that the GPU implementation is accurate enough for practical use cases and the performance measurements made on synthetic datasets are representative.

6. Related Work

Cross-correlation relates to the problem of signal processing in many different fields and we have collected several examples where the GPU processing creates an edge. In the domain of radio astronomy, all signals from radio antennas need to be usually correlated with each other, which puts this problem in the HPC domain. Various cross-correlation optimizations have been proposed: Clark et al. [3] developed a GPU kernel, which promotes tiling and optimized memory transfer. By utilizing both FPGAs and GPUs, Ord et al. [10] propose a hybrid approach to achieve sufficient performance. Ragoomundun et al. [11] utilize batched matrix multiply routines of the cuBLAS GPU library to implement their optimized correlator to enable real-time processing for telescopes.

Seismic interferometry is another use case, where cross-correlation plays a major role. An increasing amount of seismometers allows the production of more detailed seismic information of the Earth but it is typically limited by the processing runtime. Zhou et al. [4] optimize noise cross-correlation functions used to obtain Earth's underground structures. Ventosa et al. [12] implement a GPU version of phase cross-correlation, which is used in Interstation correlation. Beaucé et al. [13] discuss optimizations of Fast Matched Filter, which is an important tool in the detection of seismic events.

Applications of cross-correlation can be also found in computer vision. Fan et al. discuss autonomous vehicle applications in the context of disparity maps [1] (used for stereo vision) or lane detection [14]. Typically, mobile platforms such as autonomous cars and robots have strict limits to their power intake, so Syed et al. [15] and Chang et al [16] described ways to further optimize stereo vision algorithms on embedded hardware, such as Nvidia Jetson GPU, to achieve the required speed of processing while maintaining low power consumption.

Similar examples can be found in other signal-processing domains. For instance, Belloch et al. proposed a multi-GPU implementation for acoustic localization where signals from an array of microphones need to be processed [2]. The analysis employs a traditional FFT approach (using cuFFT for the transformation) and it was accompanied by custom CUDA kernels for the remaining operations. The interesting part is that multiple signals need to be processed simultaneously.

It has been established that fast cross-correlation is useful in various practical domains. However, most of the papers mentioned in the previous put little effort into the optimizations of the algorithm and provide only straightforward GPU implementations. We would like to introduce also several works which have influenced our proposed solution. Perhaps the most fundamental is the well-known BLAS library called Magma [17]. It is one of the first libraries that effectively utilized two-level tile caching (shared memory and registers) in matrix multiplication.

Similar caching can be employed when image tiles are being compared many times. An example of an algorithm that relies heavily on comparing image tiles is Block-matching and 3D denoising, which has a very efficient CUDA implementation by Honzátko et al. [18]. Similarly to cross-correlation, the BM3D algorithm

searches for similarity between image parts, so it compares different overlapping tiles. In the CUDA implementation, the authors made an observation that the overlapping work can be computed only once and re-used. They also employed an efficient work distribution pattern where an entire warp cooperates on a comparison of a single patch. Closer to our research, a CUDA-accelerated implementation of 3D stereo vision [19] employs cross-correlation computed on neighborhoods of all pixels to determine relative shifts between images taken from stereo cameras. The implementation of the 3D vision was quite efficient thanks to effective caching in shared memory, albeit it was implemented for a rather specific Nvidia Jetson TX2 device.

We found no elaborate optimizations directly for the cross-correlation, but more thorough research was done in the domain of convolution, especially in methods related to training neural networks. Yan et al. [20] presented an optimized GPU implementation for batched Winograd convolutions. Similarly to us, they have observed the low arithmetic density of their solution and attempted to mitigate the problem by cleverly caching the data in the registers. The solution presented by Lu et al. [21] introduces even more complex optimizations. In particular, they employ warp-wise buffers managed by warp-shuffle instructions and data reuse patterns similar (but simpler) to our grouped-overlap optimization. However, the convolution algorithms optimize for larger input on one side and rather small filters on the other side, so it is not directly applicable for general cross-correlation.

The work that inspired our design probably the most was the CUDA implementation of Levenshtein's edit distance [22]. It uses circular warp-wise buffers and clever utilization of warp-shuffle instructions that lead to a very efficient algorithm that is quite fast despite the unavoidable data dependencies inherent to the Levenshtein. It also uses double buffering to promote coalesced loads, similar to our left-matrix buffers.

Finally, there is one aspect of modern GPUs that we have not focused on in our work. Contemporary NVIDIA architectures since Volta incorporate *Tensor units* in the GPU streaming multiprocessors. These units are specifically designed to perform fused multiply-add instructions (FMA), which are essential in many computations including cross-correlation. The tricky part is to use them efficiently since they are designed only for particular combinations of FMAs that are used in neural networks. Kikuchi et al. [23] presented an implementation specifically tailored for the use of CUDA tensor cores. They employ *Warp Matrix Multiply-Accumulate API* to compute multiple waveform pairs with multiple shifts (overlaps) simultaneously. The solution is claimed to achieve better performance than cuBLAS, but it is applicable only for 1D cross-correlation. A similar idea was proposed by Yamaguchi et al. [24] earlier, but they have focused on half-precision (FP16) computations. The FMA optimizations were omitted from our paper for the sake of brevity, but they definitely present another possibility to achieve even better performance.

7. Conclusions

We have proposed a novel approach to definition-based implementation of cross-correlation for contemporary GPUs. The proposed algorithm takes advantage of the data reuse principle — i.e., the operations are rearranged so that every value loaded into a register is used multiple times. This way, the load operations from global memory are reduced significantly, which leads to overall performance improvement. To extend this idea further, we designed a data-exchange schema where the values in registers are shuffled among neighboring threads using warp-shuffle instructions, which are much faster than loads from global memory and measurably faster than shared memory. We have also experimented with different scenarios when multiple (shared) input matrices are cross-correlated simultaneously, which enables another level of parallelism and data reuse. The optimizations presented in this paper can lead to a speedup that exceeds an order of magnitude with respect to naïve (baseline) CUDA implementation.

We have also compared our algorithms with a traditional FFT approach. As expected, in the case of small matrices, the definition-based approach significantly outperforms the cuFFT implementation due to the costly initialization and preprocessing phase of the FFT transform. In the case of *one-to-one* correlation, the *warp-shuffle* algorithm is better even for 256×256 matrices. When multiple matrices are correlated (the *n-to-m* scenario), the turning point is roughly at the size of 64×64. The proposed algorithms are also available (along with many other implementations we experimented with) as source codes provided in the attached replication package, so our conclusions may be independently verified and the code may be easily adapted for immediate application.

Acknowledgements

This paper was supported by Charles University institutional funding SVV 260698/2023.

References

[1] R. Fan, N. Dahnoun, Real-time implementation of stereo vision based on optimised normalised cross-correlation and propagated search range on a gpu, in: 2017 IEEE International Conference on Imaging Systems and Techniques (IST), IEEE, 2017, pp. 1–6.

[2] J. A. Belloch, A. Gonzalez, A. M. Vidal, M. Cobos, On the performance of multi-gpu-based expert systems for acoustic localization involving massive microphone arrays, Expert Systems with Applications 42 (13) (2015) 5607–5620.

[3] M. A. Clark, P. C. L. Plante, L. J. Greenhill, Accelerating radio astronomy cross-correlation with graphics processing units (Jul. 2011). arXiv:1107.4264.

[4] J. Zhou, Q. Wei, C. Wu, G. Sun, A high performance computing method for noise cross-correlation functions of seismic data, in: 2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), IEEE, 2021, pp. 1179–1182.

[5] M. A. Medina, P. Schwille, Fluorescence correlation spectroscopy for the detection and study of single molecules in biology, Bioessays 24 (8) (2002) 758–764.

[6] K. Kapinchev, A. Bradu, F. Barnes, A. Podoleanu, Gpu implementation of cross-correlation for image generation in real time, IEEE, Cairns, QLD, Australia, 2015, pp. 1–6. doi:10.1109/ICSPCS.2015.7391783.

[7] A. Šmelko, M. Kruliš, K. Maděra, Asociated GitHub repository with source code and experimental data (2024).
URL `https://github.com/asmelko/jpdc23-artifact`

[8] L. Zhang, T. Wang, Z. Jiang, Q. Kemao, Y. Liu, Z. Liu, L. Tang, S. Dong, High accuracy digital image correlation powered by gpu-based parallel computing, Optics and Lasers in Engineering 69 (2015) 7–12. doi:https://doi.org/10.1016/j.optlaseng.2015.01.012.
URL `https://www.sciencedirect.com/science/article/pii/S0143816615000135`

[9] R. Bracewell, P. B. Kahn, The fourier transform and its applications, American Journal of Physics 34 (8) (1966) 712–712.

[10] S. M. Ord, B. Crosse, D. Emrich, D. Pallot, R. B. Wayth, M. A. Clark, S. E. Tremblay, W. Arcus, D. Barnes, M. Bell, et al., The murchison widefield array correlator, Publications of the Astronomical Society of Australia 32 (2015) e006.

[11] N. Ragoomundun, G. Beeharry, A cublas-based gpu correlation engine for a low-frequency radio telescope, Astronomy and Computing 32 (2020) 100407.

[12] S. Ventosa, M. Schimmel, E. Stutzmann, Towards the processing of large data volumes with phase cross-correlation, Seismological Research Letters 90 (4) (2019) 1663–1669.

[13] E. Beaucé, W. B. Frank, A. Romanenko, Fast Matched Filter (FMF): An Efficient Seismic Matched-Filter Search for Both CPU and GPU Architectures, Seismological Research Letters 89 (1) (2017) 165–172. arXiv:https://pubs.geoscienceworld.org/ssa/srl/article-pdf/89/1/165/4018649/srl-2017181.1.pdf, doi:10.1785/0220170181.
URL `https://doi.org/10.1785/0220170181`

[14] R. Fan, N. Dahnoun, Real-time stereo vision-based lane detection system, Measurement Science and Technology 29 (7) (2018) 074005.

[15] I. A. Syed, M. Datar, S. Patkar, Accelerated stereo vision using nvidia jetson and intel avx, in: Computer Vision and Image Processing: 5th International Conference, CVIP 2020, Prayagraj, India, December 4-6, 2020, Revised Selected Papers, Part II 5, Springer, 2021, pp. 137–148.

[16] Q. Chang, A. Zha, W. Wang, X. Liu, M. Onishi, L. Lei, M. J. Er, T. Maruyama, Efficient stereo matching on embedded gpus with zero-means cross correlation, Journal of Systems Architecture 123 (2022) 102366.

[17] S. Tomov, R. Nath, P. Du, J. Dongarra, Magma users' guide, ICL, UTK (November 2009) (2011).

[18] D. Honzátko, M. Kruliš, Accelerating block-matching and 3d filtering method for image denoising on GPUs, Journal of Real-Time Image Processing 16 (6) (2017) 2273–2287. doi:10.1007/s11554-017-0737-9.

[19] H. Cui, N. Dahnoun, Real-time stereo vision implementation on nvidia jetson tx2, in: 2019 8th Mediterranean Conference on Embedded Computing (MECO), 2019, pp. 1–5. doi:10.1109/MECO.2019.8760027.

[20] D. Yan, W. Wang, X. Chu, Optimizing batched winograd convolution on gpus, in: Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming, 2020, pp. 32–44.

[21] G. Lu, W. Zhang, Z. Wang, Optimizing depthwise separable convolution operations on gpus, IEEE Transactions on Parallel and Distributed Systems 33 (1) (2021) 70–87.

[22] D. Bednárek, M. Brabec, M. Kruliš, Improving matrix-based dynamic programming on massively parallel accelerators, Information Systems 64 (2017) 175–193. doi:10.1016/j.is.2016.06.001.

[23] Y. Kikuchi, K. Fujita, T. Ichimura, M. Hori, L. Maddegedara, Calculation of cross-correlation function accelerated by tensor cores with tensorfloat-32 precision on ampere gpu, in: International Conference on Computational Science, Springer, 2022, pp. 277–290.

[24] T. Yamaguchi, T. Ichimura, K. Fujita, A. Kato, S. Nakagawa, Matched filtering accelerated by tensor cores on volta gpus with improved accuracy using half-precision variables, IEEE Signal Processing Letters 26 (12) (2019) 1857–1861. doi:10.1109/LSP.2019.2951305.

Contribution 4
MaBoSS

Published as Adam Šmelko et al. "Maboss for HPC environments: implementations of the continuous time Boolean model simulator for large CPU clusters and GPU accelerators". *BMC bioinformatics* **25** (2024)

BMC Bioinformatics

SOFTWARE

Open Access

Maboss for HPC environments: implementations of the continuous time Boolean model simulator for large CPU clusters and GPU accelerators

Adam Šmelko[1], Miroslav Kratochvíl[2], Emmanuel Barillot[3,4,5] and Vincent Noël[3,4,5*]

*Correspondence:
vincent.noel@curie.fr

[1] Department of Distributed and Dependable Systems, Charles University, Prague, Czech Republic
[2] Luxembourg Centre for Systems Biomedicine, University of Luxembourg, Esch-sur-Alzette, Luxembourg
[3] Institut Curie, Université PSL, 75005 Paris, France
[4] INSERM, U900, 75005 Paris, France
[5] Mines ParisTech, Université PSL, 75005 Paris, France

Abstract

Background: Computational models in systems biology are becoming more important with the advancement of experimental techniques to query the mechanistic details responsible for leading to phenotypes of interest. In particular, Boolean models are well fit to describe the complexity of signaling networks while being simple enough to scale to a very large number of components. With the advance of Boolean model inference techniques, the field is transforming from an artisanal way of building models of moderate size to a more automatized one, leading to very large models. In this context, adapting the simulation software for such increases in complexity is crucial.

Results: We present two new developments in the continuous time Boolean simulators: MaBoSS.MPI, a parallel implementation of MaBoSS which can exploit the computational power of very large CPU clusters, and MaBoSS.GPU, which can use GPU accelerators to perform these simulations.

Conclusion: These implementations enable simulation and exploration of the behavior of very large models, thus becoming a valuable analysis tool for the systems biology community.

Keywords: Computational biology, High performance computing, Boolean models

Introduction

Biological systems are large and complex, and understanding their internal behavior remains critical for designing new therapies for complex diseases such as cancer. A crucial approach in this endeavor is building computational models from existing knowledge and analyzing them to find intervention points and to predict the efficacy of new treatments [1, 2]. Many different frameworks have been used to describe biological systems, from quantitative systems of differential equations to more qualitative approaches such as Boolean models [3]. While the former seems more adapted to represent complex behavior, such as non-linear dependencies, the latter is being increasingly used because

of its capability to analyze very large systems. Many Boolean models have been built to describe biological systems to tackle a variety of problems: from understanding fundamental properties of cell cycle [4, 5] to advanced properties of cancer [6–8].

Historically, the task of building Boolean models involved reading an extensive amount of literature and summarizing it in a list of essential components and their interactions. More recently, database listings of such interactions [9, 10] and experimental information retrieval techniques on a bigger number of components were subjected to many advancements. Combined with the design of automatic methods for Boolean formulae inference from the constraints encoded in the knowledge and the experimental data [11–14], these new developments allows the construction of large Boolean models. While this effort faces many challenges, we believe it is a promising way to study the large-scale complexity of biological systems. However, in order to analyze the dynamic properties of such large Boolean models, we need to develop efficiently scalable simulation tools.

Here, we present adaptations of MaBoSS [15, 16]—a stochastic Boolean simulator that performs estimations of state probability trajectories based on Gillespie stochastic simulation algorithm [17]—to modern HPC computing architectures, which provide significant speedups of the computation, thus allowing scrutinization and analysis of much larger Boolean models. In particular, the problem of properly quantifying low abundant phenotypes [18] can now be tackled by making more realistic the large number of simulation needed to cover the space of possible trajectories. The main contributions comprise two new implementations of MaBoSS:

- MaBoSS.GPU, a GPU-accelerated implementation of MaBoSS, which is designed to exploit the computational power of massively parallel GPU hardware.
- MaBoSS.MPI, a parallel implementation of MaBoSS which can scale to multinode environments, such as large CPU clusters.

The source code of the proposed implementations is publicly available at their respective GitHub repositories.[1] We also provide the scripts, presented plots, data and instructions to reproduce the benchmarks in the replication package.[2]

To showcase the utility of the new implementations, we performed benchmarking on both existing models and large-scale synthetic models. As the main results, MaBoSS.GPU provided over 200× speedup over the current version of MaBoSS on a wide range of models using contemporary GPU accelerators, and MaBoSS.MPI is capable of almost linear performance scaling with added HPC resources, allowing similar speedups by utilizing the current HPC infrastructures.

Background

Boolean signaling models

A Boolean signaling model consists of n *nodes*, which can represent a gene, protein or an event in a cell. Nodes are either active or inactive, gaining binary values 1 or 0

[1] https://github.com/sysbio-curie/MaBoSS.GPU, https://github.com/sysbio-curie/MaBoSS.

[2] https://github.com/sysbio-curie/hpcmaboss-artifact.

respectively. The *state* of the whole model is represented by a vector S of n binary values where S_i represents the value of the i-th node. We denote the set of all possible states as $\mathcal{S} = \{0,1\}^n$; thus $|\mathcal{S}| = 2^n$.

Interactions in the model are described as transitions between two states. A single state can have multiple transitions to other states with assigned transition probabilities. In turn, a Boolean network is represented as a directed weighted graph $G = (\mathcal{S}, \rho)$, where $\rho : \mathcal{S} \times \mathcal{S} \to [0, \infty)$ is a transition function generating *transition rates*. These rates define edge weights of G, which are used to compute the probability of a transition from state S to S' in the following way:

$$P(S \to S') = \frac{\rho(S, S')}{\sum_{S'' \in \mathcal{S}} \rho(S, S'')}. \tag{1}$$

For convenience, it holds that

$$\rho(S, S') = 0 \iff \text{there is no transition from } S \text{ to } S'. \tag{2}$$

MaBoSS: Markovian Boolean stochastic simulator

MaBoSS simulates the *asynchronous update strategy*, where only a single node changes its value in each transition (as opposed to the *synchronous update strategy*, for which all nodes that can be updated are updated [15]). Therefore, there is a transition from S to S' only if it holds that

$$\begin{aligned} &S_j \neq S'_j \text{ for a given } j \\ &S_i = S'_i \text{ for } i \neq j. \end{aligned} \tag{3}$$

Consequently, S can have at most n possible transitions. In programming terms, S' is obtained by flipping the j-th bit of S.

To determine the possible transition rates, each node follows the *Boolean logic* $\mathcal{B}_i : \mathcal{S} \to [0, \infty)$, which determines the expected Poisson-process rate of transitioning to the other value. If $\mathcal{B}_i(S) = 0$, then the transition at node i is not allowed in state S. Given this formalization, the simulation can be also viewed as a continuous-time Markov process.

The main computational part of the Boolean logic is its binary function $f : \mathcal{S} \to \{0,1\}$, which consists of logical operators (such as *and*, *or*, *xor*, *not*) with nodes as operands. For example,

$$f_i(S) = (S_2 \wedge S_3) \vee S_4 \tag{4}$$

is the binary function for node i, having nodes 2, 3 and 4 as its operands. The binary function of a node determines the value to which the node can transition. Thus, S can transition at node i at rate r only if $f_i(S) \neq S_i$. Concisely, $\mathcal{B}_i$ is defined as[3]

[3] Generally, $\mathcal{B}_i$ can be defined using a pair of binary formulas [15]: one used when a node is active and one when it is inactive. This results in a slightly more branched definition, which we omitted for brevity.

$$\mathcal{B}_i(S) = \begin{cases} 0 & \text{if } S_i = f_i(S) \\ r & \text{otherwise} \end{cases} \tag{5}$$

MaBoSS algorithm simulates the above process to produce stochastic *trajectories*: sequences of states $S^0, S^1, \ldots, S^k$ and time points $t^0 < t^1 < \cdots < t^k$ where $t^0 = 0$ and S^0 is the initial state, and for each $i \in \{0, \ldots, k-1\}$, S^i transitions to S^{i+1} at time t^{i+1}. The simulation ends either by a timeout when reaching the maximal allowed time, or by reaching a *fixed point* state with no outgoing transitions. The algorithm for a single iteration of the trajectory simulation is given explicitly in Algorithm 1, which is the direct application of the Gillespie stochastic simulation algorithm on the Boolean state space.

Algorithm 1 A single iteration of the MaBoSS simulation of a trajectory, given the state *S* and time *t*.

1: **procedure** TRAJECTORYSIMULATIONSTEP(S, t)
2: $\rho_1 \leftarrow \mathcal{B}_1(S), \ldots, \rho_n \leftarrow \mathcal{B}_n(S)$ ▷ compute transition rates
3: $r \leftarrow \text{random}([0, \sum_{i=1}^{n} \rho_i))$
4: $i \leftarrow \min_i \sum_{j=1}^{i-1} \rho_j \leq r < \sum_{j=1}^{i} \rho_j$ ▷ randomly select the transition node
5: $S' \leftarrow S$ with the i-th bit flipped
6: $u \leftarrow \text{random}([0, 1])$
7: $\delta t \leftarrow -\frac{\ln u}{\sum_{i=1}^{n} \rho_i}$ ▷ randomly select time to the next transition
8: **return** $(S', t + \delta t)$
9: **end procedure**

Multiple trajectories are generated and aggregated in compound trajectory statistics. Commonly obtained statistics include:

- *Network state probabilities on a time window*—Trajectory states are divided by their transition times into time windows based on the time intervals specified by a window size. For each window, the probability of each state is computed as the duration spent in the state divided by the window size. The probabilities of the corresponding windows are then averaged across all subtrajectories.
- *Final states*—The last sampled states from the trajectories are used to compute a final state distribution.
- *Fixed states*—All reached fixed points are used to compute a fixed state distribution.

To maintain the brevity in the statistics, MaBoSS additionally allows marking some nodes *internal*. This is useful because nodes that are not "interesting" from the point of final result view occur quite frequently in Boolean models, and removing them from statistics computation often saves a significant amount of resources.

Computational complexity of parallel MaBoSS algorithm

Simulation complexity

We estimate the time required to simulate c trajectories as follows: For simplification, we assume that a typical Boolean logic formula in a model of n nodes can be evaluated in $\mathcal{O}(n)$ (this is a very optimistic but empirically valid estimate). With that, the computation of all possible transition rates (Algorithm 1, line 2) can be finished in $\mathcal{O}(n^2)$. The

selection of the flipping bit (Algorithm 1, line 4) can be finished in $\mathcal{O}(n)$, and all other parts of the iteration can finish in $\mathcal{O}(1)$. In total, the time complexity of one iteration is $\mathcal{O}(n^2)$. If we simulate c trajectories with an upper bound of trajectory length u, the simulation time is in $\mathcal{O}(c \cdot u \cdot n^2)$.

In an idealized PRAM (parallel random access machine [19]) model with infinite parallelism, we can optimize the algorithm in the following ways:

- Given c processors, all trajectory simulations can be performed in parallel, reducing the time complexity to $\mathcal{O}(u \cdot n^2)$. (Note that this does not include the results aggregation. See *Statistics aggregation* section for further description.)
- With n processors, the computation of transition rates in the simulation can be done $\mathcal{O}(n)$ time, and the selection of the flipping bit can be done in $\mathcal{O}(\log n)$ time using a parallel prefix sum, giving $\mathcal{O}(n)$ time for a single iteration.

Thus, using a perfect parallel machine with $c \cdot n$ processors, the computation time can be reduced to $\mathcal{O}(u \cdot n)$. Notably, the $\mathcal{O}(u)$ simulation steps that must be performed serially remain a major factor in the whole computation time.

Statistics aggregation

The aggregation of the statistics from the simulations is typically done by updating a shared associative structure indexed by model states, differing only in update frequency between the three kinds of collected statistics.

If the associative structure is implemented as a hashmap, the updates can be done in $\mathcal{O}(1)$ for a single process. With multiple processors, the algorithm may hold partial versions of the hashmap for each processor, and aggregate all of them at the end of the computation, which can be done in $\mathcal{O}(\log c \cdot m)$ using c processors, assuming the maximal size of statistic to be m.

As an interesting detail, the hash structures pose a surprising constant-factor overhead. In networks where most nodes are internal, the hash map may be replaced by a fixed-size multidimensional array that holds an element for all possible combinations of external node values (basically forming a multidimensional histogram). We discuss the impact of this optimization in *Implementation* section.

MaBoSS CPU implementation

MaBoSS was initially developed as a single-core application, but swiftly, it was extended with a basic parallelism to exploit the multi-core nature of modern CPUs. In this parallel implementation, the simulation of trajectories and the statistics aggregation were distributed among multiple cores using POSIX threads. In the following sections of the papers, this implementation will serve as a baseline, and we will refer to it simply as the *CPU version*.

Each statistics data held by a thread is represented by a hash map with the keys as the states of the model and the values as a numerical value. Therefore, their aggregation from multiple trajectories of multiple threads is carried out by a well-researched parallel sum reduction. To better understand how a researcher can use MaBoSS output, in the

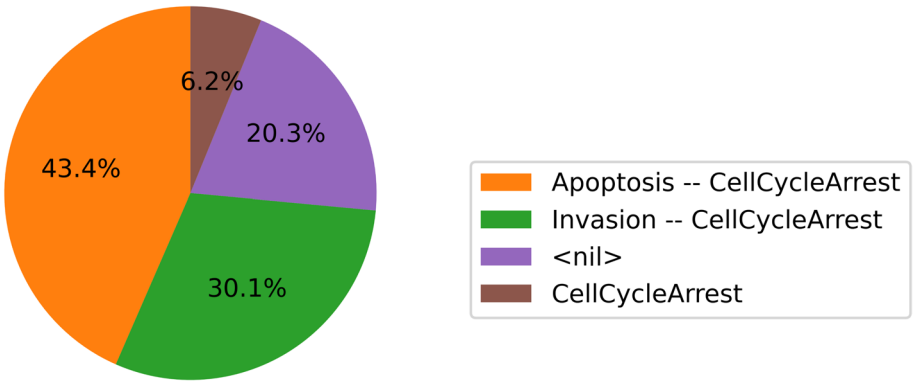


Fig. 1 The final states pie chart shows the distribution of the last trajectory states. Labels denote which active non-internal nodes compose the state. *nil* label represents the state where all non-internal nodes are inactive

following section, we discuss the differences between the statistics in greater detail and show the standard ways of their visualization.

Statistics output and visualization

Each of the three kinds of statistics is in its nature a sample from a probabilistic distribution of Boolean states. This sample is represented in code as a hash map in the CPU version, varying in the (*key*, *value*) pairs according to the specific statistic. For the final states, the keys of the hash map are the model states, and the values are the number of times the state was sampled as the last in a trajectory. Such output can be visualized as a pie chart (see Fig. 1). The fixed states are represented similarly, but only the fixed points are stored in the hash map as keys.

The final and fixed state statistics characterize the behavior of the model at one point in time—at the end of the simulation. The network state probabilities on a time window highlight more dynamic characteristics of the model, showing how the average trajectory evolves over the simulation time. Programmatically, it is an extension of the final state statistics—instead of one hash map, there is a hash map for each time window. The hash map values are the state durations in the specific time window aggregated over all simulated trajectories. Further, these statistics can be visualized in various ways using a line chart. Figure 2 shows which non-internal nodes are active throughout the simulation.

As mentioned at the beginning of the section, if the trajectory does not reach a fixed point, the simulation is stopped after the maximal allowed time. This is a common scenario, especially when some trajectories form cycles, i.e., when a model has *cyclic attractor*, also known as limit cycles. A limit cycle is usually not directly visible from the state probability line charts; Stoll et al. [15] proposed methods to detect them (such as plotting the state and transition entropies), but we do not discuss the methods further in this paper for the sake of brevity.

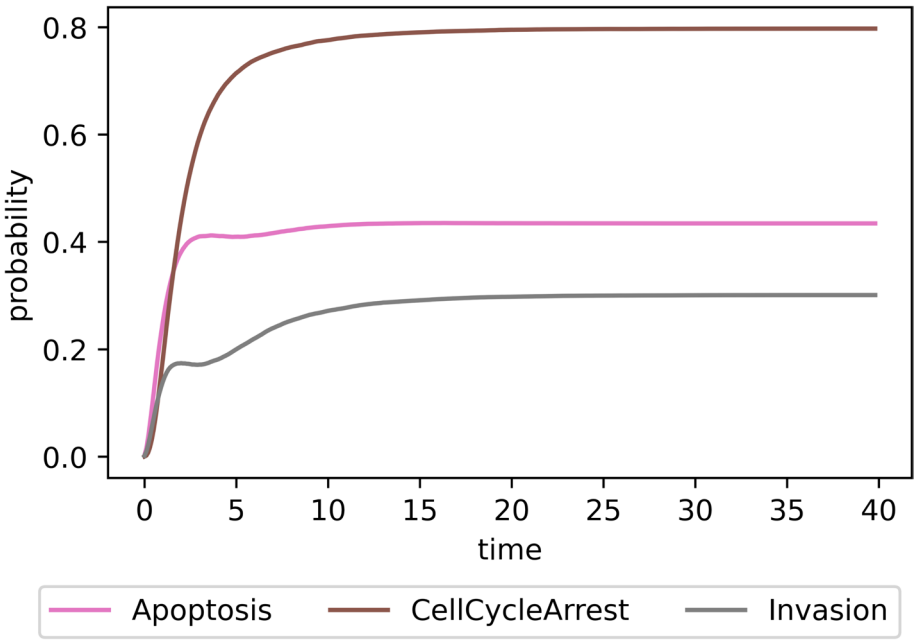


Fig. 2 The line chart of trajectory state probabilities over time windows. Each line represents the ratio of an active non-internal node in the time window over all trajectories (e.g., at the beginning of the simulation, the *Apoptosis* node is inactive in all simulated trajectories and as the time reaches the value of 10, *Apoptosis* is active in around 40% of trajectories). The x-axis represents the discrete simulation time with the window width of 0.1

Implementation

MaBoSS.GPU

Simulation

In the CPU version of MaBoSS, the simulation part is the most computationally demanding part, with up to 80% of MaBoSS runtime spent by just evaluating the Boolean formulae (the exact number depends on the model). The original formula evaluation algorithm in MaBoSS used a recursive traversal of the expression tree, which (apart from other issues) causes memory usage patterns unsuitable for GPUs: the memory required per each core is not achievable in current GPUs, and there are typically too many cache misses [20].

There are multiple ways to optimize the expression trees for GPUs: One may use a linked data structure that is more cache-friendly such as the van Emde Boas tree layout [21], or perhaps represent the Boolean formulae as a compact continuous array, or convert it to CNF or DNF (conjunctive or disjunctive normal form) bitmasks that can be easily evaluated by vector instructions. We decided to leave the exact representation choice on the compiler, by encoding the expressions as direct code and using the runtime compilation of GPU code [22]. In such an approach, the application reads the model files, writes the formulae as functions in CUDA C++ language, compiles them using the NVIDIA runtime compiler, and finally runs the simulation on GPU—all without user intervention.

Using this technique, the Boolean formulae are compiled as functions into a native binary code, which is directly executed by the GPU. As the main advantage, the formulae are encoded in the instructions, preventing unnecessary fetches of the encoded formulae from other memory. At the same time, the compiler may apply a vast spectrum of optimizations on the Boolean formulae, including case analysis and shortcutting, again resulting in faster evaluation.

A possible drawback of the runtime compilation stems from the relative slowness of the compiler—for small models, the total execution time of MaBoSS.GPU may be easily dominated by the compilation.

The work distribution was chosen to be one trajectory simulation per GPU thread. Due to the involved implementation complexity, we avoided optimization of the computation of individual trajectories by splitting the Boolean function evaluation into multiple threads (thus missing the factor of n threads from the asymptotic analysis). While such optimization might alleviate some cache pressure and thus provide significant performance improvements, we leave its exploration to future work.

Statistics aggregation

For optimizing the statistics aggregation, MaBoSS.GPU heavily relies on the fact that the typical number of non-internal nodes in a real-world MaBoSS model rarely exceeds 10 nodes, regardless of the size of the model. This relatively low number of states generated by non-internal nodes allows us to materialize the whole statistics structure (called "histogram") as a fixed-size array (rarely exceeding 2^{10} elements).

This approach allows us to avoid storing the states as the keys and gives a simple approach that can map the state to the histogram index using simple bit masking and shifting instructions. Further, we use several well-known GPU histogram update optimizations to improve the performance, including shared memory privatization and atomic operations.

MaBoSS.MPI

MaBoSS.MPI is a straightforward extension of the original MaBoSS CPU code to the MPI programming interface. Briefly, each MPI node is assigned to simulate the same number of trajectories (up to a remainder). These are further uniformly distributed among the CPU cores of the node, each thread progressively collecting the results into a privatized hashmap-based statistics aggregation structure.

Once all trajectory simulations are finished and the statistics are computed for each thread, the intermediate data are reduced into the final result using MPI collective operations.

Results

To evaluate the impact of the implemented optimizations, we present the results of performance benchmarks for MaBoSS.GPU and MaBoSS.MPI by comparing their runtimes against the original CPU implementation. To obtain a comprehensive overview of achievable results, we used both real-world models and synthetic models with varying sizes.

Table 1 The main features of the synthetic and real-world models used in the benchmarks.

Model	# Nodes (non-inter.)	# Traj.	Avg. formula size	Avg. traj. length
CELLCYCLE	10 (4)	1M	4	26
SIZEK	87 (4)	1M	22	525
MONTAGUD	133 (3)	1M	4	197
Synthetic	10–1000 (5)	1–100M	10–100	100

It includes the size of models in terms of nodes and non-internal nodes, the number of simulated trajectories, the average formula size measured as the arithmetic mean of operands count in each formula, and the average length of all simulated trajectories. Note that not all combinations of features for the synthetic model were used in the benchmarks, see the following figures for more details

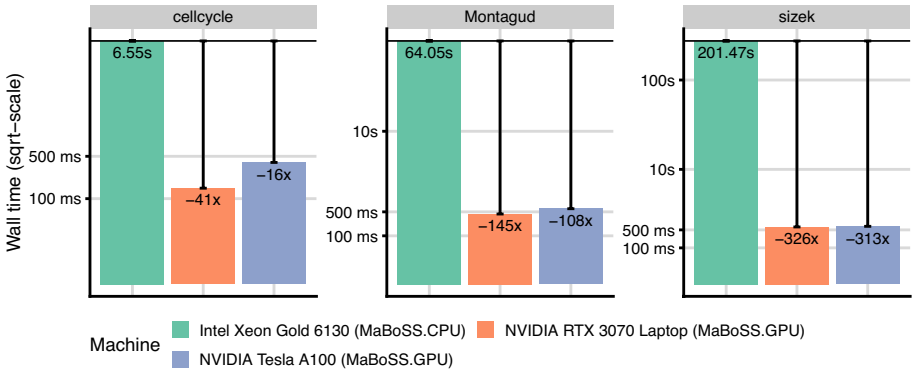


Fig. 3 Wall time comparison of MaBoSS and MaBoSS.GPU on real-world models. Each model is simulated with 1 million trajectories

Benchmarking methodology

For the benchmarks, we used 3 real-world models of 10, 87 and 133 nodes (CELLCYCLE [4], SIZEK [5] and MONTAGUD [8]). In order to test the scalability of the GPU and MPI implementation, we also created several synthetic models with up to 1000 nodes. Synthetic models were designed in a way such that the length of each simulated trajectory is predictable, and the models have no stable states. The average length was arbitrarily set to 100, which creates reasonably-sized serial tasks to saturate the tested hardware well. Also, the number of non-internal nodes was kept low (5 nodes) to enable the usage of the histogram optimization. The synthetic models together with their Python generator are available in the replication package. Table 1 summarizes the main features of the benchmarked models.

The GPU implementation benchmarks were run on a datacenter-grade NVIDIA Tesla A100 GPU and a consumer-grade NVIDIA RTX 3070 Laptop GPU. The CPU implementation benchmarks were run on a 32-core Intel Xeon Gold 6130 CPU with multithreading. The CPU implementation was compiled with GCC 13.2.0, and the GPU implementation was compiled with CUDA 12.2. Each measurement was repeated 10 times, and the average runtime was used as the final result.

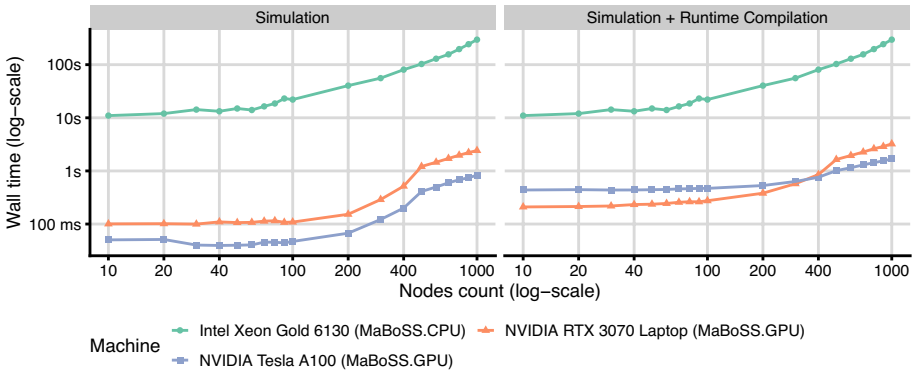


Fig. 4 Wall time comparison of MaBoSS and MaBoSS.GPU on synthetic models with sizes ranging from 10 to 1000 nodes (x-axis) and the formula size of 10. Each model is simulated with 1 million trajectories. The two panels differ by the inclusion of the runtime compilation of the model logic, showing its impact on total run time

The MPI implementation benchmarks were run on the MareNostrum 4 supercomputer.[4]

Performance of MaBoSS.GPU

In Fig. 3, we compare the wall time of the CPU and GPU implementations on real-world datasets. The GPU implementation is faster than the CPU implementation on all models, and the speedup shows to be more significant on the models with more nodes and longer trajectories. On the MONTAGUD model with 133 nodes, but a relatively short average trajectory, we achieve 145× speedup. On a slightly smaller SIZEK model with a longer average trajectory, the speedup is up to 326×.

It is worth noting that the datacenter GPU performs worse than the laptop GPU. Both devices are bottlenecked by the runtime compilation of the Boolean formulae, however, NVIDIA A100 spends on average around 300ms more on the compilation step. Subtracting the compilation time, A100 is faster for all models. We did not spend time finding the root cause of this discrepancy since the value is negligible and the following benchmarks show that the runtime compilation overhead quickly disappears with increasing model size.

Figure 4 shows much finer performance progression on synthetic models. We observed that the CPU variant starts to progress steeper at around the 100 nodes boundary. We assume that the implementation hits the cache size limit, and the overhead of fetching the required data from the memory becomes dominant. The same can be observed in the GPU variant later at around 200 nodes. Expectably, the cache-spilling performance penalty is much more significant on GPUs. Overall, the results suggest that the optimization of dividing transition rate computations among multiple threads, as mentioned in *Implementation* section, may provide a better speedup for bigger models, as it alleviates the register and cache pressure.

Additionally, Fig. 4 shows the total runtime of the GPU implementation including the runtime compilation step. Comparing the panels, we observe that the relative runtime

[4] https://www.bsc.es/marenostrum/marenostrum

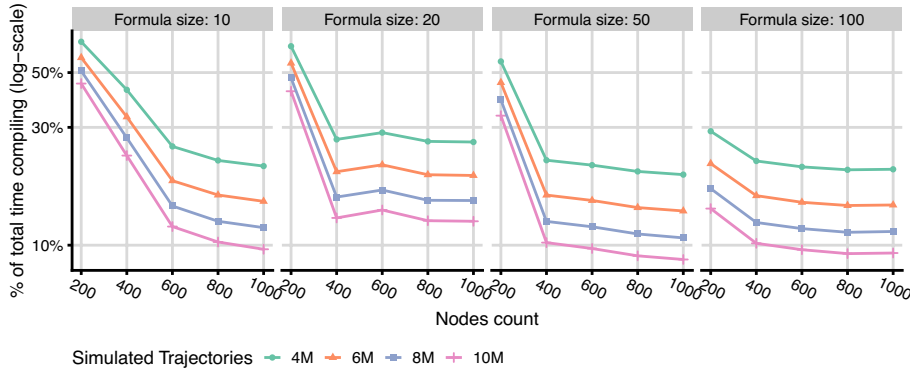


Fig. 5 The ratio of time spent in the runtime compilation of the Boolean formulae in relation to the total runtime, simulating models with varying numbers of nodes, trajectories, and formula lengths

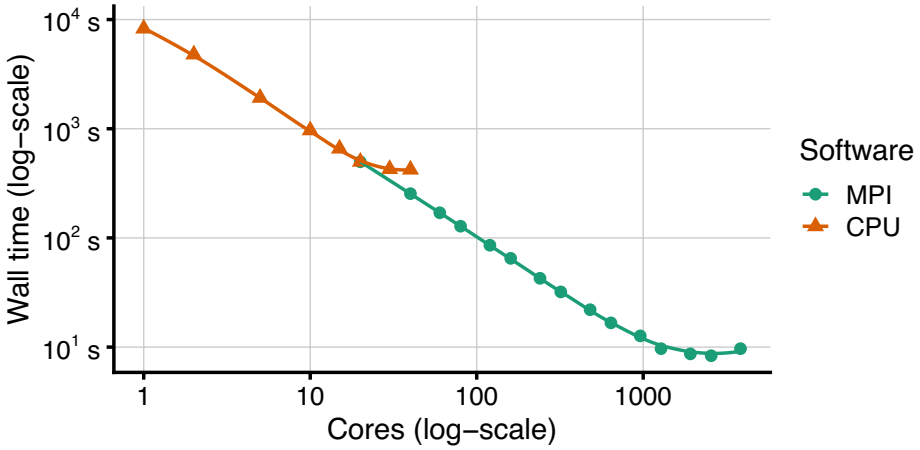


Fig. 6 Scalability results of MPI implementation on Sizek model simulating 1 million trajectories with up to 192 MPI nodes and 20 cores per node, summing up to 3840 cores

compilation overhead quickly disappears with increasing model size. Figure 5 shows the results of more detailed benchmarks for this scenario, as run on the NVIDIA Tesla A100 GPU. We observed that the compilation time is linearly dependent on the number of nodes and formula lengths, which can be simply explained by the fact that these model properties extend source files that need to be compiled by a linear factor. Notably, as soon as the simulation becomes more computationally complex (e.g., by increasing the number of nodes, the number of simulated trajectories or their average length), the compilation time becomes relatively negligible even for models with unrealistically long formulae. This suggests that the runtime compilation is a viable optimization methodology also for much larger models.

Performance of MaBoSS.MPI

Figure 6 shows the efficiency of the MaBoSS.MPI implementation on the SIZEK model. We ran multiple suites, ranging from a single MPI node up to 192 nodes, each running 20 cores. We can observe a close-to-linear speedup of up to 64 MPI nodes (1280 cores),

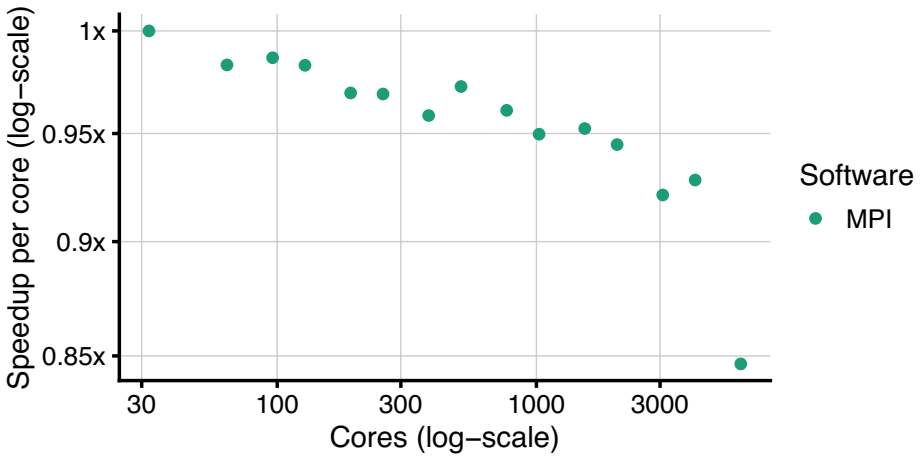


Fig. 7 Speedup scaling of MPI implementation on the synthetic model with 1000 nodes, 100 million trajectories and the formula size of 10, running on up to 192 MPI nodes with 32 cores per MPI node, summing up to 6144 cores

and a plateau for larger suites (Fig. 6, green). This can be explained by hitting an expectable bottleneck in parallelization overhead and MPI communication cost when the problem is divided into too many small parts.

To stress the scalability of the implementation, we also used the synthetic model with 1000 nodes running 100 million trajectories. We simulated this model on 32 cores per MPI node, on 1 to 192 nodes (32 to 6144 cores). The obtained speedups are summarized in Fig. 7. Using this configuration, the simulation time decreases from 20 h on 1 MPI node to 430 s on 192 nodes. As expected, the plateau in the speedup was observed only for much bigger suites. More specifically, we can see a pronounced decrease in the speedup at 192 nodes, hitting the aforementioned bottleneck during the utilization of more than 4096 cores.

Conclusions

In this work, we presented two new implementations of MaBoSS tool, a continuous time Boolean model simulator, both of which are designed to enable utilization of the HPC computing resources: MaBoSS.GPU is designed to exploit the computational power of massively parallel GPU hardware, and MaBoSS.MPI enables MaBoSS to scale to many nodes of HPC clusters via the MPI framework. We evaluated the performance of these implementations on real-world and synthetic models and demonstrated that both variants are capable of providing significant speedups over the original CPU code. The GPU implementation shows 145–326× speedup on real-world models, and the MPI implementation delivers a close-to-linear strong scaling on big models.

Overall, we believe that the new MaBoSS implementations enable simulation and exploration of the behavior of very large, automatically generated models, thus becoming a valuable analysis tool for the systems biology community.

Future work

During the development, we identified several optimization directions that could be taken by researchers to further scale up the MaBoSS simulation approach.

Mainly, the parallelization scheme used in MaBoSS.GPU could be enhanced to also parallelize over the evaluation of Boolean formulae. To avoid GPU thread divergence, this would however require a specialized Boolean formula representation, entirely different from the current version of MaBoSS; likely even denying the relative efficiency of the use of runtime compilation. On the other hand, this optimization might decrease the register pressure created by holding the state data, and thus increase the performance on models with thousands of nodes.

In the long term, easier optimization paths might lead to sufficiently good results: For example, backporting the GPU implementation improvements back to the MaBoSS CPU implementation could improve the performance even on systems where GPU accelerators are not available. Similarly, both MaBoSS.GPU and MaBoSS.MPI could be combined into a single software that executes distributed GPU-based analysis over multiple MPI nodes, giving a single high-performance solution for extremely large problems.

Availability and requirements

- Project name: MaBoSS.GPU
- Project home page: https://github.com/sysbio-curie/MaBoSS.GPU
- Operating system(s): Platform independent
- Programming language: C++, CUDA
- Other requirements: Flex, Bison, CMake >= 3.18, Cuda toolkit >= 12.0
- License: MIT
- Any restrictions to use by non-academics: None
- Project name: MaBoSS.MPI
- Project home page: https://github.com/sysbio-curie/MaBoSS
- Operating system(s): Platform independent
- Programming language: C++
- Other requirements: Flex, Bison
- License: BSD3-clause
- Any restrictions to use by non-academics: None

Abbreviations

HPC	High performance computing
CPU	Central processing unit
GPU	Graphical processing unit
MPI	Message passing interface
PRAM	Parallel random access machine
CNF	Conjunctive normal form
DNF	Disjunctive normal form

Acknowledgements
We thank Laurence Calzone and Gautier Stoll for their guidance and fruitful discussions.

Author contributions
A.S. implemented MaBoSS.GPU, V.N. implemented MaBoSS.MPI. M.K., E.B., V.N. supervised the project. All authors wrote the manuscript. All authors reviewed the manuscript.

Funding
The research leading to these results has received funding from the European Union's Horizon 2020 Programme under the PerMedCoE Project (http://www.permedcoe.eu), grant agreement n^{o} 951773. The project was partially supported by Charles University, SVV project number 260698.

Availability of data and materials
The scripts, presented plots, data and instructions to reproduce the benchmarks are available on GitHub [23].

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no conflict of interest.

Received: 22 March 2024 Accepted: 20 May 2024
Published online: 24 May 2024

References

1. Brodland GW. How computational models can help unlock biological systems. In: Seminars in cell and developmental biology, vol 47. Elsevier; 2015. pp. 62–73.
2. Bongrand P. Understanding how cells probe the world: a preliminary step towards modeling cell behavior? Int J Mol Sci. 2023;24(3):2266.
3. Machado D, Costa RS, Rocha M, Ferreira EC, Tidor B, Rocha I. Modeling formalisms in systems biology. AMB Express. 2011;1:1–14.
4. Fauré A, Naldi A, Chaouiya C, Thieffry D. Dynamical analysis of a generic Boolean model for the control of the mammalian cell cycle. Bioinformatics. 2006;22(14):124–31. https://doi.org/10.1093/bioinformatics/btl210.
5. Sizek H, Hamel A, Deritei D, Campbell S, Ravasz Regan E. Boolean model of growth signaling, cell cycle and apoptosis predicts the molecular mechanism of aberrant cell cycle progression driven by hyperactive pi3k. PLoS Comput Biol. 2019;15(3):1006402. https://doi.org/10.1371/journal.pcbi.1006402.
6. Fumiã HF, Martins ML. Boolean network model for cancer pathways: predicting carcinogenesis and targeted therapy outcomes. PLoS ONE. 2013;8(7):1–11. https://doi.org/10.1371/journal.pone.0069008.
7. Wooten DJ, Groves SM, Tyson DR, Liu Q, Lim JS, Albert R, Lopez CF, Sage J, Quaranta V. Systems-level network modeling of small cell lung cancer subtypes identifies master regulators and destabilizers. PLoS Comput Biol. 2019;15(10):1–29. https://doi.org/10.1371/journal.pcbi.1007343.
8. Montagud A, Béal J, Tobalina L, Traynard P, Subramanian V, Szalai B, Alföldi R, Puskás L, Valencia A, Barillot E, Saez-Rodriguez J, Calzone L. Patient-specific Boolean models of signalling networks guide personalised treatments. Elife. 2022;11:72626. https://doi.org/10.7554/eLife.72626.
9. Licata L, Lo Surdo P, Iannuccelli M, Palma A, Micarelli E, Perfetto L, Peluso D, Calderone A, Castagnoli L, Cesareni G. Signor 2.0, the signaling network open resource 2.0: 2019 update. Nucleic Acids Res. 2020;48(D1):504–10. https://doi.org/10.1093/nar/gkz949.
10. Türei D, Korcsmáros T, Saez-Rodriguez J. Omnipath: guidelines and gateway for literature-curated signaling pathway resources. Nat Methods. 2016;13(12):966–7. https://doi.org/10.1038/nmeth.4077.
11. Aghamiri SS, Singh V, Naldi A, Helikar T, Soliman S, Niarakis A. Automated inference of Boolean models from molecular interaction maps using CaSQ. Bioinformatics. 2020;36(16):4473–82. https://doi.org/10.1093/bioinformatics/btaa484.
12. Chevalier S, Noël V, Calzone L, Zinovyev A, Paulevé L. Synthesis and simulation of ensembles of Boolean networks for cell fate decision. In: Computational methods in systems biology: 18th international conference, CMSB 2020, Konstanz, Germany, September 23–25, 2020, Proceedings, vol 18. Springer; 2020. pp. 193–209. https://doi.org/10.1007/978-3-030-60327-4_11.
13. Beneš N, Brim L, Huvar O, Pastva S, Šafránek D. Boolean network sketches: a unifying framework for logical model inference. Bioinformatics. 2023;39(4):158. https://doi.org/10.1093/bioinformatics/btad158.
14. Prugger M, Einkemmer L, Beik SP, Wasdin PT, Harris LA, Lopez CF. Unsupervised logic-based mechanism inference for network-driven biological processes. PLoS Comput Biol. 2021;17(6):1–30. https://doi.org/10.1371/journal.pcbi.1009035.
15. Stoll G, Viara E, Barillot E, Calzone L. Continuous time Boolean modeling for biological signaling: application of Gillespie algorithm. BMC Syst Biol. 2012;6(1):1–18. https://doi.org/10.1186/1752-0509-6-116.
16. Stoll G, Caron B, Viara E, Dugourd A, Zinovyev A, Naldi A, Kroemer G, Barillot E, Calzone L. Maboss 2.0: an environment for stochastic Boolean modeling. Bioinformatics. 2017;33(14):2226–8. https://doi.org/10.1093/bioinformatics/btx123.
17. Gillespie DT. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. J Comput Phys. 1976;22(4):403–34.
18. Gillespie DT, Hellander A, Petzold LR. Perspective: stochastic algorithms for chemical kinetics. J Cheml Phys. 2013;138:17.
19. Fortune S, Wyllie J. Parallelism in random access machines. In: Proceedings of the tenth annual ACM symposium on theory of computing; 1978. pp. 114–118.

20. Karlsson M, Dahlgren F, Stenstrom P. A prefetching technique for irregular accesses to linked data structures. In: Proceedings sixth international symposium on high-performance computer architecture. HPCA-6 (Cat. No. PR00550). IEEE; 2000. pp. 206–217. https://doi.org/10.1109/HPCA.2000.824351.
21. Emde Boas P. Preserving order in a forest in less than logarithmic time. In: 16th Annual symposium on foundations of computer science (sfcs 1975). IEEE; 1975. pp. 75–84. https://doi.org/10.1016/0020-0190(77)90031-X.
22. CUDA NVRTC. 2023. https://docs.nvidia.com/cuda/nvrtc/index.html. Accessed 14 Feb 2024.
23. Šmelko A. sysbio-curie/hpcmaboss-artifact: updated scripts, plots, data and instructions to reproduce the benchmarks for the MaBoSS HPC paper. https://doi.org/10.5281/zenodo.11128107.

Publisher's Note

Contribution 5
Noarr Layouts

Published as Adam Šmelko et al. “Astute Approach to Handling Memory Layouts of Regular Data Structures”. In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2022, pp. 507–528

Astute Approach to Handling Memory Layouts of Regular Data Structures

Adam Šmelko[1], Martin Kruliš[1(✉)], Miroslav Kratochvíl[2], Jiří Klepl[1], Jiří Mayer[1], and Petr Šimůnek[1]

[1] Department of Distributed and Dependable Systems, Charles University, Prague, Czech Republic
{smelko,krulis}@d3s.mff.cuni.cz
[2] Luxembourg Centre for Systems Biomedicine, University of Luxembourg, Esch-sur-Alzette, Luxembourg
miroslav.kratochvil@uni.lu

Abstract. Programmers of high-performance applications face many challenging aspects of contemporary hardware architectures. One of the critical aspects is the efficiency of memory operations which is affected not only by the hardware parameters such as memory throughput or cache latency but also by the data-access patterns, which may influence the utilization of the hardware, such as re-usability of the cached data or coalesced data transactions. Therefore, a performance of an algorithm can be highly impacted by the layout of its data structures or the order of data processing which may translate into a more or less optimal sequence of memory operations. These effects are even more pronounced on highly-parallel platforms, such as GPUs, which often employ specific execution models (lock-step) or memory models (shared memory).

In this work, we propose a modern, astute approach for managing and implementing memory layouts with first-class structures that is very efficient and straightforward. This approach was implemented in Noarr, a GPU-ready portable C++ library that utilizes generic programming, functional design, and compile-time computations to allow the programmer to specify and compose data structure layouts declaratively while minimizing the indexing and coding overhead. We describe the main principles on code examples and present a performance evaluation that verifies our claims regarding its efficiency.

Keywords: Memory layout · Data structure · Cache · Parallel · Performance · Reusable

1 Introduction

This paper aims to tackle memory-related performance issues, which represent one of the most crucial performance optimization topics. In hardware, memory access is optimized by providing faster memories closer to the chip (like HBM2), multi-level caches and transfer buffers, and even specialized explicit near-core

W. Meng et al. (Eds.): ICA3PP 2022, LNCS 13777, pp. 507–528, 2023.
https://doi.org/10.1007/978-3-031-22677-9_27

memories (such as AVX512 registers or shared memory in Nvidia GPUs). Software developers benefit from these features by creating specialized, cache-aware algorithms, often tailored for a particular architecture.

The design of the way that the program data is laid out in memory is one of the crucial steps that ensures memory access performance. Even simple design choices like row- or column-major matrix storage impact the performance within the complex memory cache models by simplifying address translations, improving cache hit ratio and prefetching, or ensuring the alignment required for coalesced SIMD operations [7,14]. For parallel algorithms, the complexity of the problem becomes much broader because of cache-line collisions, false-sharing, non-uniform memory architectures, a variety of synchronization issues [3,11,18], and other factors. Many-core platforms (GPUs in particular) only amplify this by enforcing specific data access patterns in lockstep execution, advocating the use of programmer-managed caches (like shared memory), and having a significantly lower cache-to-core ratio in comparison to the CPUs [13].

The best layout is quite often elusive and needs to be discovered empirically. Furthermore, it often differs even among the utilized cache levels [10,12,17]. Consequently, the optimal implementations are often complicated, and most of the optimization-relevant code is not portable between hardware architectures. Enabling simple implementations of layout-flexible data structures and algorithms would improve the code portability (and value); however, systematic approaches are quite rare, often over-complicating the code logic and making the algorithm implementation not maintainable or usable beyond the community of specialists.

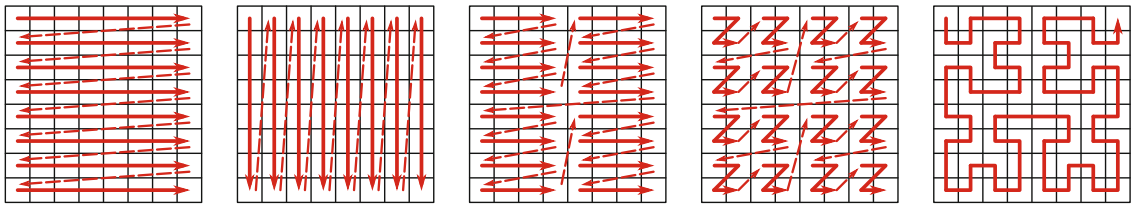
1.1 Motivational Example

To explain the motivation, objectives, and contributions of our research, we have selected a matrix multiplication problem widely known in computer science. For the sake of simplicity, we use the most straightforward implementation with $\mathcal{O}(N^3)$ complexity (computing $C = A \times B$ of square matrices N^2):

```
for (size_t i = 0; i < N; ++i)
    for (size_t j = 0; j < N; ++j) {
        C[i][j] = 0;
        for (size_t k = 0; k < N; ++k)
            C[i][j] += A[i][k] * B[k][j];
    }
```

Having a fixed algorithm structure (i.e., order of the operations), the memory layout of the matrices is the main issue affecting the performance. In this context, the layout is defined by transforming the abstract indices (i, j) into an offset, subsequently used to compute the actual memory address. For instance, the most common matrix layout is row-major, which computes the offset as $i*W+j$ (where W is the width of the matrix). A few examples of possible layouts are depicted in Fig. 1.

The aforementioned code sample used traditional C notation `A[i][j]` which enforces the row-major layout, which is sub-optimal for this algorithm. Having the second matrix in a col-major layout or using a z-curve for all matrices will



(a) row-major (b) col-major (c) row-tiles (d) z-curve (e) Hilbert curve

Fig. 1. Examples of common matrix layouts

improve cache utilization, and the algorithm would run several times to several orders of magnitude faster, depending on the platform. Therefore, we need to introduce layout flexibility into the code.

A typical object-oriented solution would be to create a class abstraction that would define a uniform interface for accessing matrix elements whilst enabling different implementations through derived classes. A slightly better and more reusable solution would be to separate the offset computation into a policy class that would be injected into the matrix as a template parameter:

```
class RowMajor {
    static size_t offset(size_t i, size_t j, size_t W, size_t H) {
        return i*W + j;
    }
};

template<typename T = float, class Layout = RowMajor>
class Matrix {
    /* ... */
    T& at(size_t i, size_t j) {
        return _data[Layout::offset(i, j, _W, _H)];
    }
};
```

The policy class makes the matrix implementation flexible (in terms of selecting the proper layout) and efficient (since the compiler can inline the static method). However, several drawbacks make this solution imperfect. The interface between the `Matrix` class and its layout policy (`RowMajor`) is created ad-hoc by the author of the main class, which complicates the code reusability of the layout policies in potentially compatible situations. The interface also prevents efficient constant propagation and caching of intermediate values. Furthermore, the strong encapsulation may prevent low-level optimizations, portability to other architectures (e.g., GPUs), and complicate data structure composition (e.g., when matrices in an array need to be interleaved).

We aim to design a more straightforward, more programmer-friendly solution to implementing *layout-agnostic* algorithms, focusing on enabling performance optimizations and parallel processing.

1.2 Objectives and Contributions

Our main objective was to create a library that allows the users to quickly adapt their algorithms and data structures for different memory layouts, with a particular focus on the following targets:

- Once an algorithm is adapted, it becomes layout-agnostic—i.e., no subsequent internal code modifications should be required to change the layout of the underlying data structures.
- The layout representation should not be coupled with memory allocation so that it could be used in different scenarios and different memory spaces (i.e., directly applicable with memory-mapped files or GPU unified memory).
- The interface should define an easily comprehensible abstraction for *indexing* (offset computation) that would hide its (possibly complex) nuances.
- The indexing mechanism should enable the compiler to evaluate constant expressions at compile time (e.g., fold constant dimensions of a structure into the generated code).
- The code overhead should be minimal, preferably smaller than with well-established practices, such as providing template policy classes to govern layout or allocation.

We have implemented Noarr header-only library[1] for C++ as a prototype that achieves the outlined objectives. C++ was chosen as a widely-used mainstream language that provides complete control over memory layout and allocation and is widely used for programming performance-critical applications, including parallel HPC systems and GPGPU computing. Its fundamental features, like the templating system and operator overloading, open possibilities for generic programming, compile-time optimizations, and the design of a functional-like interface, which simplifies the use of the library. Furthermore, the separation of indexing from (CPU-specific) memory management allowed us to directly utilize the library with Nvidia CUDA code, easily porting the layout-agnostic code on contemporary GPUs.

We believe that Noarr will make a significant contribution to simplifying the coding process and increasing performance in many scenarios, especially:

- Empirical exploration of possible layouts—i.e., finding the optimal combination of layouts for given data structures and algorithms by measuring the performance of all possible implementations.
- Implementing applications and libraries in which the optimal layout of data structures needs to be selected at runtime (e.g., based on the size of the problem or the best available architecture).
- Allowing simple yet efficient (semi)automatic layout transformations in case the input or output layouts differ from the optimal layouts for the computation.

Although the issues mentioned above can be identified in a large variety of data structures and algorithms, we are focusing mainly on regular data structures such as nested multi-dimensional arrays and structures (in the C/C++ sense). However, despite this narrow scope, we have identified that this problem is quite challenging, especially regarding optimizations for massively parallel environments like GPUs.

[1] Noarr is available as open-source on GitHub under MIT license: https://github.com/ParaCoToUl/noarr-structures.

The paper is organized as follows. Section 2 explains the key principles and benefits of the layout-agnostic algorithm design. The performance aspects of offset computation overhead are summarized in Sect. 3. In Sect. 4, we provide insights into the current implementation of the Noarr library. Related work and main conclusions are summarized in Sects. 5 and 6.

2 Extensible Memory Layout Structures

One of the most significant challenges of the outlined problem is to create an indexing abstraction that would follow the fundamental code design principles (especially in object-oriented programming, which is one of the most widely adopted paradigms), thus allowing the programmer to write neat and maintainable code, whilst minimizing performance overhead and making heavy use of the compile-time optimizations.

In this work, we propose using first-class indexing structures which can be detached entirely from the allocated memory and the data structures themselves. The indexing structure has a specific type (templated class) composed of predefined base types and a corresponding instance (object). This way, the information being passed to the layout-agnostic algorithm is divided into two parts:

- the data type passed via (inferred) template parameter, which bears the structure and constant parameters,
- and the object, which bears all dynamic parameters (such as sizes of non-constant dimensions of the data structure).

Before we focus on the benefits, let us emphasize the C++ cornerstones of Noarr that are pretty important for understanding the main principles (details are provided in Sect. 4).

- The indexing structure type composition is straightforward as the user merely combines predefined Noarr templated classes. Furthermore, thanks to the templating system, it is easy to create partially-defined structures, thus promoting code reusability. The construction of derived or augmented types (like binding the constant dimensions) is implemented in a functional manner, which is quite comprehensive and easy to write. Finally, modern C++ constructs like `auto` or template type inference make these type modifications easier to handle since only the instance object is passed down.
- The dimensions of the data structure are denoted using chars (typically letters), which are much more mnemonic than numbers or the order of definition. Furthermore, they can be used to define additional abstraction so that structures with the same set of named dimensions can be treated as compatible, regardless of the order of their definition or their actual layout representation.
- Finally, the implementation makes heavy use of `constexpr` functions which allow the compiler to be inlined, resolve, and even precompute many pieces of the layout-related code, thus making it more efficient. For instance, the

```
template <char I, char J, class struct_lhs_t, class struct_rhs_t, class struct_out_t>
__global__ float matmul_tile(const float* lhs_in, const float* rhs_in, float* out, const
↪  struct_lhs_t lhs_s, const struct_rhs_t rhs_s, struct_out_t out_s) {
    constexpr size_t tile_w = 16;
    constexpr auto tile_s = noarr::array<I, tile_w, noarr::array<J, tile_w,
    ↪  noarr::scalar<float>>>();
    __shared__ float l_tile[tile_w * tile_w];
    __shared__ float r_tile[tile_w * tile_w];
    const uint32_t x = blockIdx.x * tile_size + threadIdx.x;
    const uint32_t y = blockIdx.y * tile_size + threadIdx.y;

    float acc = 0.f;
    for (uint32_t i = 0; i < lhs_s.get_length<J>(); i += tile_w) {
        tile_s.get_at<I, J>(l_tile, threadIdx.y, threadIdx.x) =
            lhs_s.get_at<I, J>(lhs_data, y, threadIdx.x + i);
        tile_s.get_at<I, J>(r_tile, threadIdx.y, threadIdx.x) =
            rhs_s.get_at<I, J>(rhs_data, threadIdx.y + i, x);
        __syncthreads();

        for (uint32_t j = 0; j < tile_w; j++)
            acc += tile_s.get_at<I, J>(l_tile, threadIdx.y, j)
                    * tile_s.get_at<J, I>(r_tile, threadIdx.x, j);
        __syncthreads();
    }
    out_s.get_at<I, J>(output_data, y, x) = acc;
}
```

Listing 1: CUDA matrix multiplication kernel based on Noarr library

constant dimensions can be translated into the expressions where the actual memory offsets are being computed, which may allow optimizations like precomputing constant subexpressions.

Utilizing memory layouts as first-class objects can introduce some flexibility into the code. In this section, we demonstrate the two main ideas of the proposed approach: The ability to easily *decouple memory allocation from its interpreted layout* and the possibility of writing *memory-layout-agnostic functions*. Listing 1 presents an example that employs both these ideas using Noarr library.

2.1 Decoupling the Memory Management

In C++, memory is usually acquired following one of two scenarios—either it is allocated internally by a wrapping data structure (the 'owning' semantics), or it is provided by the caller (the 'borrowing' semantics). When the indexing structure is decoupled from the memory allocation and combined with the borrowing semantics, it can cover many elaborate memory management scenarios, such as file memory-mapping or sharing memory among threads (this also includes CUDA unified memory or shared memory).

In Noarr, the layout objects are entirely independent of memory management. To simplify the situation for programmers, it also provides a wrapper structure `bag`, which binds the layout structure with any pointer, acting as a smart pointer with borrowing semantics. The layout can be used alone to compute linearized offsets from input indices, which is also applicable in hypothetical scenarios beyond pointer-based memory addressing.

We present an example of a matrix multiplication kernel implemented in CUDA (Listing 1) to demonstrate the possibilities opened by proper decoupling. In the code, a GPU kernel performs the multiplication in tiles where each 16×16 tile of the output matrix is computed by one thread block, and each element is handled by one thread. A thread block cooperatively fetches a pair of tiles from the input matrices (one pair at a time) into the shared memory; all threads of the block then use the cached tiles to update their intermediate scalar products (which are kept in their registers) before iteratively loading successive pairs of tiles. Once all tiles are processed, each thread writes its aggregated result into the output matrix.

The example focuses on a typical pattern in GPU programming—a manual caching of data in the *shared memory*. Unlike global memory (accessible by all threads), the shared memory is an integral component of a streaming multiprocessor; thus, it is dedicated to the threads within the same thread block. Unsurprisingly, the two types of memory are allocated and managed in slightly different ways, albeit both use pointer-based addressing. The global memory is usually allocated before the execution of a kernel (i.e., by the host) and passed to a kernel as an argument (`lhs_in`, `rhs_in`, and `out` on line 2 of Listing 1). The shared memory is acquired inside the kernel by defining a C array with `__shared__` prefix (`l_tile` and `r_tile` on lines 5–6).

Considering also the host memory (where a copy of matrices also needs to reside), the programmer must manage three (partial) copies in three different memory spaces. A uniform abstraction (that supports owning and borrowing semantics) streamlines the code significantly. Furthermore, in this particular instance, we could also take advantage of having a different layout for different matrices—e.g., the optimum is reached if the left-side matrix is in the row-major while the right-side matrix is in the column-major format.

Listing 1 demonstrates, how the problem is solved using Noarr. The tiles are loaded into the shared memory on lines 12–15. The variables `lhs_s` and `rhs_s` represent the layout objects, which are bound with global memory pointers (`lhs_in` and `rhs_in` respectively) to read data from input matrices (lines 13 and 15). Another layout object `tile_s` is used for two shared memory pointers representing the cached tiles (lines 12 and 14). With these layout objects, different types of memory could be accessed using the same interface. Additionally, the code is ready for future layouts modifications and promotes the reusability of the existing layout structures.

2.2 Layout-Agnostic Functions

Formally, we may define the layout-agnostic property as a unique form of polymorphism. Layout-agnostic functions are implemented in a way that does not require altering their code when the layout of the used data structures needs to be changed. As hinted in the introduction, the layout selection may significantly affect performance. In extreme cases, the relative performance improvement achieved by optimal layout selection can reach orders of magnitude.

To demonstrate this effect, we show how the layout choice changes the performance of the matrix multiplication kernel from Listing 1, which is already written as layout-agnostic. Running the kernel with different layout configurations for each matrix is implemented by simply passing different function arguments (and corresponding template parameters, which the compiler can automatically infer in typical cases). We utilize this flexibility to find a layout combination that exhibits the best performance quickly.

For the sake of this example, we coded the following matrix layouts:

- *Row-major* layout (labeled **R**, which we use as a baseline)
- *Column-major* (**C**, a transposition of row-major layout)
- ***R** tiles in **C** order* (**RC**), which divides the matrix logically into 16×16 sub-matrices (tiles); data in each sub-matrix is stored with row-major layout, while the sub-matrices are organized in column-major layout
- ***C** tiles in **R** order* (**CR**) is analogical to **RC** layout, but the tiles use column-major layout internally, and are ordered in row-major fashion
- ***CC*** and ***RR*** are defined analogically

The layout of all inputs and outputs of the matrix multiplication is thus expressed as a triplet of individual matrix layouts. For example, $\mathbf{R} \times \mathbf{C} = \mathbf{R}$ denotes a multiplication where the left and the output matrices are in row-major, and the right-side input matrix is in the column-major layout. Since the kernel 1 already caches tiles explicitly in the shared memory, we expect the tiled layouts to perform better. Likely, the $\mathbf{RR} \times \mathbf{RC} = \mathbf{R}$ should exhibit the best performance (given the properties of the algorithm).

We have created a benchmark that tested the performance of the presented algorithm using all layout combinations possible. In each test, the input matrices were loaded to the GPU global memory already transformed into the selected matrix layouts, the kernel was executed, and its execution time was measured and recorded. A relevant selection of the experimental results is shown in Fig. 2. The graphs present the normalized times (in picoseconds and femtoseconds)—i.e., kernel execution times divided by the asymptotical amount of work (N^3 in this case). Details regarding our experimental setup can be found in Appendix A, and the complete set of results can be found in our replication package[2].

The result verified that **RC** is superior to the baseline row-major layout in both input positions. Furthermore, the $R \times C = R$ configuration (often praised on sequential architectures) exhibits worse than the baseline on massively parallel hardware. While this was expected, the primary outcome of this benchmark is methodological: A selection of input and output layouts can be tested systematically without reimplementation effort, while the larger exploration size of the selection (enabled by low coding overhead) provides a solid guarantee that the best-identified solution is indeed a good choice for a high-performance software.

[2] https://github.com/asmelko/ica3pp22-artifact.

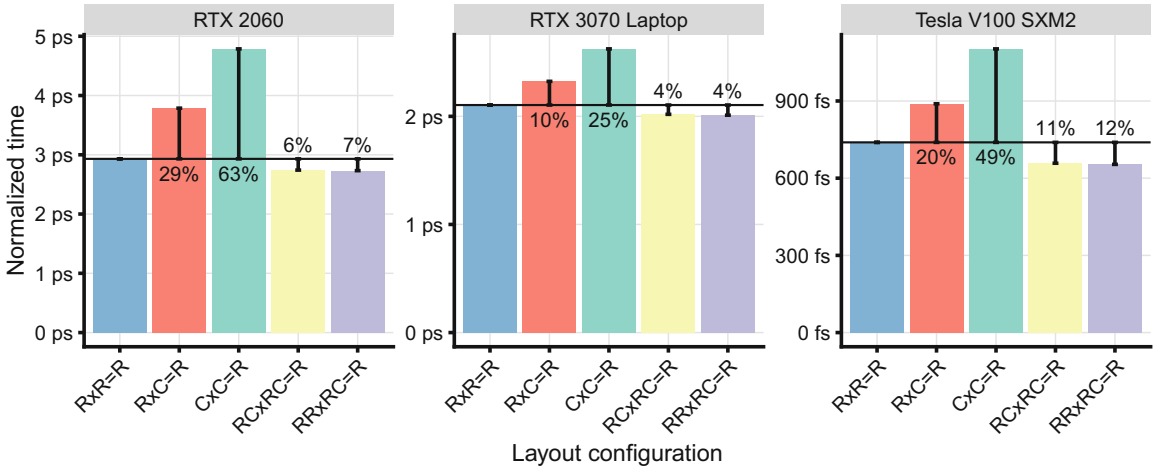


Fig. 2. Speedups of selected layout combinations relative to (row-major) baseline

```
template <char X, char Y, typename bag_in_t, typename bag_out_t>
static void transform(const bag_in_t& input_bag, bag_out_t& output_bag) {
    for (size_t i = 0; i < input_bag.get_length<X>(); i++)
        for (size_t j = 0; j < input_bag.get_length<Y>(); j++)
            output_bag.at<X, Y>(i, j) = input_bag.at<X, Y>(i, j);
}
```

Listing 2: Key part of transformation routine for 2-index (2D) arrays

2.3 Transformations

The layout-agnostic algorithms can benefit from performance gains achieved by choosing the best layout for a given problem configuration and architecture. However, in real-world scenarios, the layout of the input and output data structures is often prescribed as an inherent part of the algorithm interface or selected by the caller (in the case of generic interfaces).

If the algorithm is complex enough and the performance gap between the prescribed layouts and optimal layouts is high, the data structures may be copied and transformed into their optimally organized counterparts to speed up the algorithm. With Noarr, the transformation can be handled in a generic way. Following our examples with matrices, Listing 2 presents the central part of a generic transformer for 2D structures.

In fact, we are currently extending Noarr to handle the transformations in a generic way for any-dimensional structures, and we are exploring techniques how to select the best way of iterating the structures (e.g., selecting the best ordering of nested loops) in order to optimize memory transfers and caching. However, this research is well beyond the scope of this paper.

Transformation Overhead Assessment. Employing transformations may be beneficial only under specific circumstances. Simply put, the algorithm must save more execution time than how long it takes to transform all the necessary data.

We want to demonstrate the overhead assessment on the previously introduced matrix multiplication example.

We have analyzed the layout transformation overhead for various matrix sizes and layouts. The key results are summarized in Fig. 3. We have observed that in the case of larger matrices ($N > 10,000$), the overhead is negligible, primarily because of the asymptotic complexity difference between the transformation algorithm ($\mathcal{O}(N^2)$) and the multiplication ($\mathcal{O}(N^3)$). For smaller matrices (with N around 1000), the relative ratio of the transformation to computation time expectably increased, and the transformation overhead caused the baseline to perform the best.

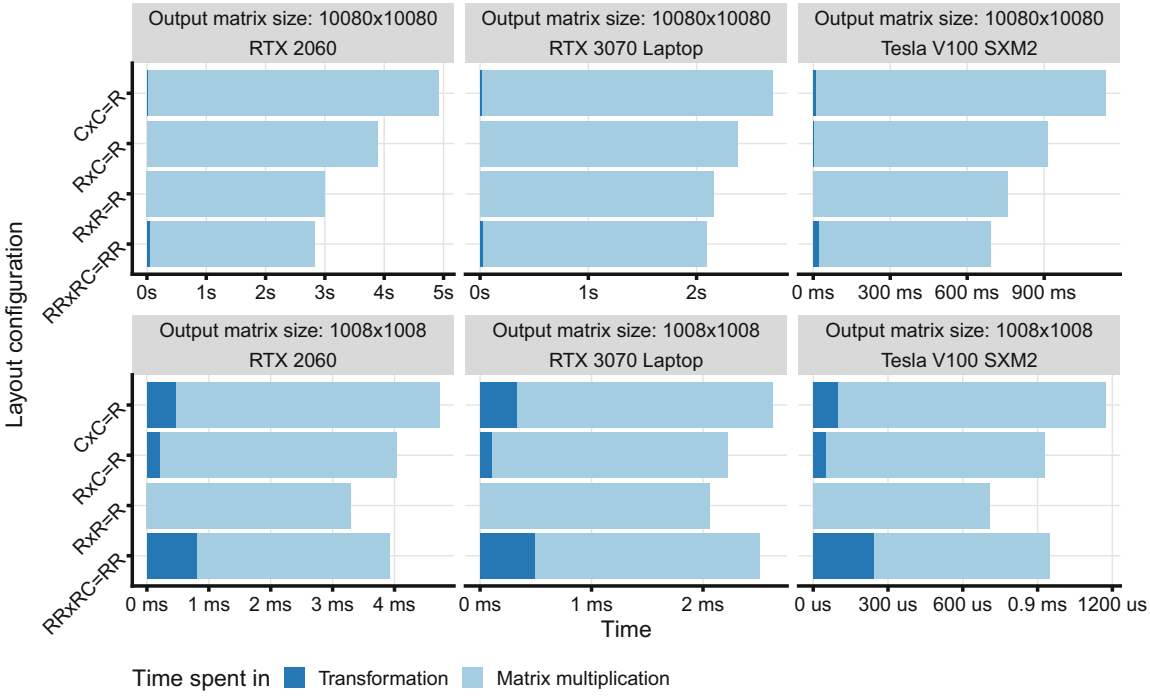


Fig. 3. Layout transformation times compared to actual matrix multiplication times

As demonstrated, deciding whether or when a layout transformation can be beneficial may be complicated; however, with Noarr, both the experiments and the actual decision to apply or not to the transformation can be implemented very quickly.

3 Performance Impact of Constant Expressions

One of the essential features of Noarr is that the first-class structures propagate along with their templated types, allowing us to embed statically defined properties (most importantly, the constant dimensions of the structure) into the type itself. Therefore, the compiler can employ optimizations like compile-time evaluation of constant expressions or exact-sized loop unrolling, which might lead

to more efficient execution or even automated vectorization. These optimizations rarely produce a game-changing improvement in performance; thus, the programmers often overlook them. However, utilization of Noarr structure will introduce them naturally so the result code could run faster without any additional effort whilst maintaining other benefits like memory allocation decoupling or coding in a layout-agnostic manner.

To present the main idea, let us have an array A of N vectors in $\mathbb{R}^D$ where N is a variable, and D is a constant[3]. We want to compute the Euclidean distance between every vector in the array and given vector q (e.g., to find k nearest vectors, which is quite a typical task in many data-processing problems):

```
for (size_t i = 0; i < N; ++i) {
  float dist = 0.0f;
  for (size_t d = 0; d < D; ++d) {
    float diff = A[i*D + d] - q[d];
    dist += diff * diff;
  }
  dist = std::sqrtf(dist); // ...
}
```

When D is a constant, the compiler could unroll the loop entirely without additional branches. It might even attempt to unroll the outer loop if D is sufficiently small. The speedup achieved by having constant D may easily reach factor $3\times$ for very small values of D (e.g., $D = 2$)[4].

3.1 Indexing Performance

To demonstrate the impact of Noarr structures, we have selected a 3D stencil problem as an example. Stencil is a simple function computed iteratively for every element of a regular grid. We have used an averaging stencil executed on a 3D grid which could be used as an approximative simulation of gas diffusion, for instance. Our objective is to emphasize the difference between situations when the grid dimensions are constant (at compile time) and when they are determined at runtime.

The main code of the stencil is in Listing 3. Run-time variables `size_x`, `size_y`, and `size_z` denote the dimensions of the cube. The first part of this experiment aims at exposing only the compile-time optimizations of index computations, so we ensure that no optimizations related to constant dimensions are performed. Please note that the loops do not visit points residing on the faces of the grid so that we can ignore the border cases of the stencil function; thus, there are no branches in the code which leads to simpler and more stable measurement.

A naïve C-like implementation of the internal `stencil` function is presented in Listing 4. It uses the same variables in the loop to index the data pointers,

[3] If the code needs to handle several different dimensionalities D, it will be compiled for each D independently thanks to the power of C++ templates.

[4] If we measure only the Euclidean distance computation.

```
template <typename... Args> void run_stencil_grid(Args&&... args) {
    for (size_t x = 1; x < size_x - 1; x++)
        for (size_t y = 1; y < size_y - 1; y++)
            for (size_t z = 1; z < size_z - 1; z++)
                stencil(std::forward<Args>(args)..., x, y, z);
}
```

Listing 3: Main stencil for-loop

preventing the compiler from doing more elaborate compile-time optimizations. This code is used as a baseline for the performance comparison.

```
inline void stencil(const float* in, float* out, size_t x, size_t y, size_t z) {
    float sum = in[x * size_y * size_z + y * size_z + z];
    sum += in[(x + 1) * size_y * size_z + y * size_z + z];
    sum += in[(x - 1) * size_y * size_z + y * size_z + z];
    sum += in[x * size_y * size_z + (y + 1) * size_z + z];
    sum += in[x * size_y * size_z + (y - 1) * size_z + z];
    sum += in[x * size_y * size_z + y * size_z + z + 1];
    sum += in[x * size_y * size_z + y * size_z + z - 1];
    out[x * size_y * size_z + y * size_z + z] = sum / 7;
}
```

Listing 4: Naïve implementation of stencil function

Making the dimensions constant may help the compiler to generate more optimal code. In C++, this can be achieved simply by defining the `size_*` variables as `constexpr`; however, such constants need to be declared at the global level, which significantly undermines any encapsulation or reusability of the code. Better way is to use fix-sized containers like `std::array` and make the stencil code templated so it can be used with any compatible containers (including `std::vector`).

```
using cube = noarr::array<'x', 1048576, noarr::array<'y', 32, noarr::array<'z', 32,
↪  noarr::scalar<float>>>>;
using bag = noarr:bag<cube, noarr::helpers::bag_policy<std::unique_ptr>>;

inline void stencil(const bag& in, bag& out, size_t x, size_t y, size_t z) {
    float sum = in.at<'x', 'y', 'z'>(x, y, z);
    sum += in.at<'x', 'y', 'z'>(x + 1, y, z);
    sum += in.at<'x', 'y', 'z'>(x - 1, y, z);
    sum += in.at<'x', 'y', 'z'>(x, y + 1, z);
    sum += in.at<'x', 'y', 'z'>(x, y - 1, z);
    sum += in.at<'x', 'y', 'z'>(x, y, z + 1);
    sum += in.at<'x', 'y', 'z'>(x, y, z - 1);
    out.at<'x', 'y', 'z'>(x, y, z) = sum / 7;
}
```

Listing 5: Noarr implementation of stencil with constant-sized `array`

Noarr provides a fixed layout structure `array`, which fulfills a similar role, but it can be easily integrated into more complex nested structures (even with custom layouts). Listing 5 presents the internal `stencil` rewritten for Noarr. The dimensions of the grid are no longer passed as variables, but they are embedded in the type of the `bag` structure as constants. Line 1 shows the assembling of the layout structure using a predefined `array` template.

To evaluate the performance, we have selected a grid of a specific size ($2^{20} \times 32 \times 32$) which confines the meaning of the diffuse simulation for a specific environment (e.g., gas in a pipe). The main reason is that the performance improvement caused by the compile-time optimizations is difficult to measure on regular structures since it takes only a small portion of overall time (especially when the computation causes many cache misses). This shape requires more index computations relative to other operations, making the difference more pronounced in the measurements.

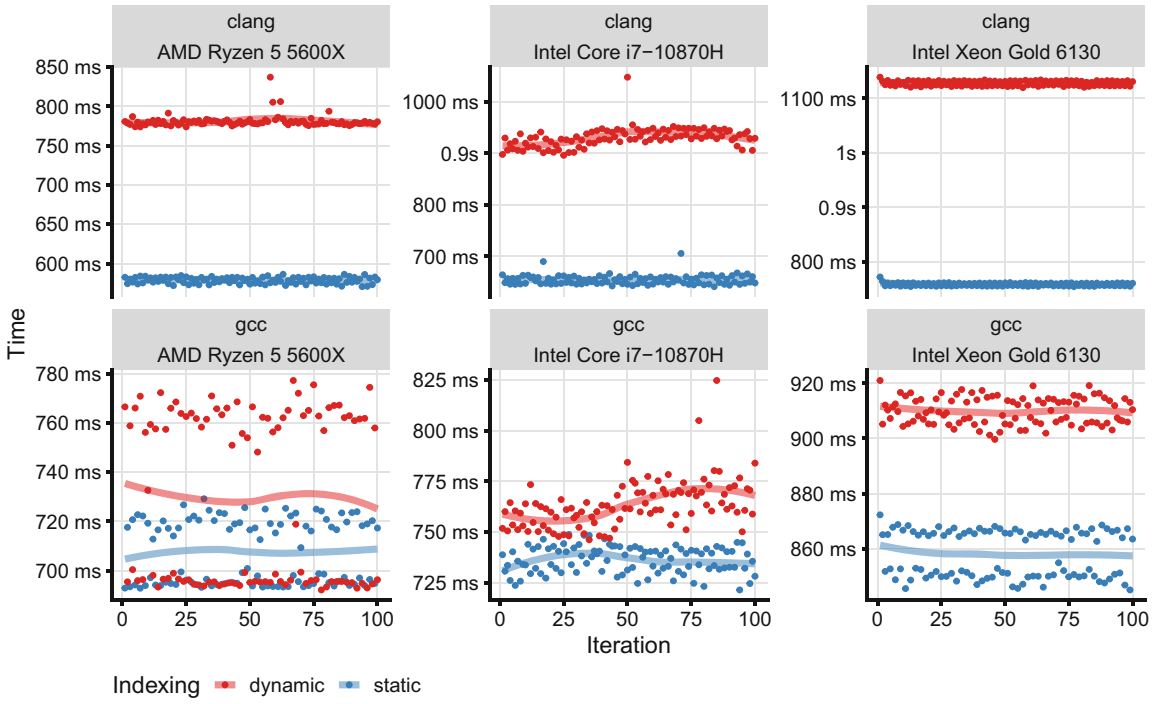


Fig. 4. Wall times of 100 stencil iterations (plotted lines represent the local regression of the measured times)

Figure 4 shows the comparison results of the two presented stencil implementations on three platforms using two compilers. The benefits of compile-time optimizations are visible on every platform and with both tested compilers, albeit there is only a small difference in some configurations. The details regarding the experimental methodology are summarized in Appendix A.

3.2 Constant-Loops Optimizations

The second part of this experiment extends the compile-time optimizations to the nested stencil grid loops. It requires replacing `size_*` variables in the main loops (Listing 3) with constants (i.e., `constexpr` or template arguments) so the compiler has enough information to perform exact loop-unrolling and better vectorization-related optimizations.

```
template <typename bag_t> constexpr void run_stencil_grid(bag_t in, bag_t out) {
    for (size_t x = 1; x < in.get_legth<'x'>() - 1; x++)
        for (size_t y = 1; y < in.get_legth<'y'>() - 1; y++)
            for (size_t z = 1; z < in.get_legth<'z'>() - 1; z++)
                stencil(in, out, x, y, z);
}
```

Listing 6: Updated stencil for-loop with `bag` structure

However, converting these variables to constants may be quite tedious, especially if we want the code to be generic for both constant and non-constant scenarios. This particular issue can be easily overcome by utilization of Noarr `bag` structures. Having the layout information encoded both in the structure type and the object, method `get_length` can query dimension sizes and returns a constant or variable based on the layout specification, all this being decided at compile time. The grid loop function from Listing 3 needs to be rewritten as demonstrated in Listing 6.

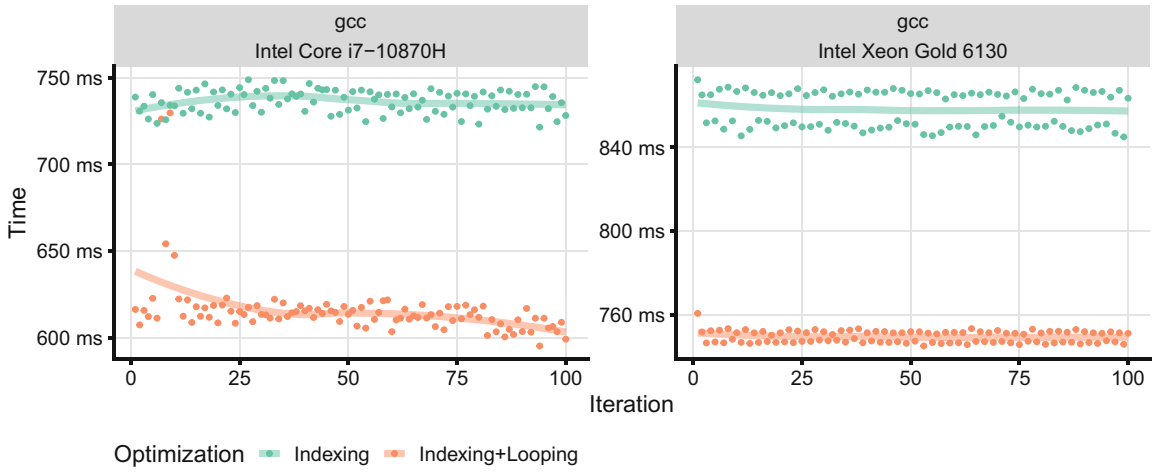


Fig. 5. Stencil execution times of two optimizations—compile-time *indexing* and the addition of constant-induced loop unrolling (*indexing+looping*)

Figure 5 presents the performance improvements of exposing constant variables to the grid iteration loop. We have included only measurements of programs compiled by `gcc` since `clang` was not able to take advantage of the constant values when they are passed through the `bag` structure interface.

4 Implementation and Technical Insights

The Noarr library[5] is logically divided into three levels, each building on top of the previous one: *structures*, *functions*, and *object wrappers*. The first two layers provide a rather low-level functional approach, while the last one encapsulates the first two into a more traditional C++ object-oriented design.

4.1 Structures

A *structure* is an object that stores information about a data layout. It exposes the information via a simple interface, providing its size in bytes (`size()`), the range of indices it supports (`length()`) and a current offset from the beginning of the structure in bytes (`offset()`).

The most trivial structure is `scalar` (Listing 7), which wraps the 'base' values to be used in more complex layouts. Scalar often wraps simple types like `float`, but it can also wrap any fixed-size C++ type (such as `struct` or `std::tuple`). The methods `length()` and `offset()` of `scalar` always return 0 because `scalar` represents only a single element.

```
template<class T>
struct scalar : contain<> {
  static constexpr size_t size() noexcept { return sizeof(T); }
  static constexpr size_t offset() noexcept { return 0; }
  static constexpr size_t length() noexcept { return 0; }
};
```

Listing 7: A core part of the `scalar` structure used for wrapping simple values

The `array` structure (Listing 8) is more complicated: Like `std::array`, it represents a fixed-size array with a named dimension and statically defined number of elements of a given *substructure* type. Unlike `scalar` which wraps a *trivial type*, `array` is contains a Noarr *structural type*.

An important aspect of the structures is their ability to be combined and nested to create a *structure tree*. For instance, the composition of `scalar` and `array` is quite straightforward:

- `array<'a', 10, scalar<float>>` defines an array of 10 `float`s,
- `array<'i', 4, array<'j', 8, scalar<int>>>` represents a 4×8 row-major integer matrix layout,
- `array<'j', 8, array<'i', 4, scalar<int>>>` represents the same matrix in a column-major layout.

[5] https://github.com/ParaCoToUl/noarr-structures.

```
1  template<char Dim, size_t L, class T>
2  struct array : contain<T> {
3    constexpr size_t size() const noexcept {
4      return contain<T>::template get<0>().size() * L;
5    }
6    constexpr size_t offset(size_t i) const noexcept {
7      return contain<T>::template get<0>().size() * i;
8    }
9    static constexpr size_t length() noexcept { return L; }
10 };
```

Listing 8: Noarr `array` structure (some methods are omitted for brevity)

All structures inherit from class `contain`, which has several purposes: It serves as recursive storage for the wrapped structure, holds some useful meta-information about the nested substructures, and stores possible additional data for the structure, such as dynamic dimension length or the current offset index. Querying for various properties, which is its main purpose, is demonstrated in Listing 8. The `array` implements the `size()` function using the information (size) from its immediate substructure (line 4). In the example, queries work recursively on subsequent immediate substructures until the recursion is halted in `scalar::size()`. Using this mechanism, `contain` allows us to create the nested hierarchy of the structure tree easily.

There are several other built-in structures in Noarr library, such as `vector` and `tuple` (analogical to `std::vector` and `std::tuple`), which provide sufficient arsenal for composing memory layouts of many regular-shaped data structures. Moreover, the library design makes it open for extensions, and programmers may implement additional custom layout structures.

4.2 Functions

Noarr *functions* are C++ `constexpr` functions that serve as an expressive tool for obtaining complex information from the structure trees. They are used to compute offsets for memory pointers to provide indexation, transform structures, and query dimension lengths using a single, extensible functional interface.

Calling function `f` on a structure `s` is achieved using the (overloaded) 'pipe' operator `|`. Expression `s | f` denotes that `f` is applied on `s` (note this may sometimes differ from `f(s)` as detailed later in this section).

For example, the function `get_length()` traverses structure tree and calls `length()` on a substructure with the given dimension name:

```
size_t i_len = a_structure | get_lenght<'i'>();
```

The function `set_length()` proceeds similarly, but when a matching substructure is found, the whole structure is reconstructed to carry the new length. The following example shows that functions can be additionally chained one after another. Notably, all structures are immutable, which allowed us to ensure that `unsized_s` does not carry any unnecessary data:

```
auto unsized_s = vector<'i', vector<'j', scalar<float>>>();
auto sized_s = unsized_s | set_length<'i'>(4) | set_length<'j'>(8);
```

A function application on a structure may fail, such as when querying a length of a non-existing dimension. We say the function is *not applicable* on a structure. Taking the aforementioned two functions into account and the fact that every structure forms a structure tree, it is possible that a function is not *directly applicable* on the topmost structure but is applicable on some structures in the structure tree. For this reason, we distinguish three *piping mechanisms* that govern different means of the function-structure application:

- *Top application* (or *direct application*). This is the simplest form of piping, where `s | f` is equivalent to `f(s)`. In other words, the function is applied directly to the topmost structure.
- *Get application.* Given the piping `s | f`, if `f(s)` is not applicable the piping mechanism attempts to apply `f` to the substructures of `s` recursively. It fails if `f` does not apply to any of the substructures or if it applies to more substructures. The trivial representative being `get_length()`, because there should be exactly one node in a structure tree with a specified dimension.
- *Transform application.* `s | f` either results in top application when `f(s)` is applicable or `f` is transformatively applied on all *direct* substructures of `s`. If the latter, the structure is reconstructed with these changes to the substructures.

The piping mechanism is implemented using C++ `constexpr` functions and metaprogramming. Together with the static nature of substructure hierarchies that encompasses the structure layer, the implementation is very efficient since it provides the necessary space for compiler optimizations. We can demonstrate this by precisely describing the operations executed when a function with the get application is applied to a structure. Let us have the following structure and function:

```
auto v4 = vector<'a', vector<'b', vector<'c', vector<'d', scalar<int>>>>>();
auto f = get_length<'d'>();
```

Expression `v4 | f` must perform a traversal of the structure tree to find the matching dimension. Fortunately, the way the structures and functions are implemented ensures that there is no run-time loop in the implementation. Because all substructures are known in compile-time, the traversal loop is unrolled using metaprogramming techniques. Furthermore, because the values are also known at compile-time, the result can be partially evaluated and, in turn, *no run-time code is generated.* In summary, applying `v4 | f` produces four unrolled function applications, three of which produce no operation at all (and usually get discarded by a compiler), and only one results in calling `length()` on a substructure that can be evaluated by the compiler.

4.3 Object Wrappers

Object wrappers provide object-oriented management of structures, functions, and the actual data. Noarr library offers two kinds of such objects—structure *wrappers* and `bags`.

A `wrapper` simplifies the work with structures by bundling the applications of the most common Noarr functions into member methods. That way, with a wrapper `w` of a structure `s` we can directly write `w.get_length<'d'>()` instead of `s | get_length<'d'>()`.

A `bag` provides the same interface as a `wrapper` but also contains a pointer to the underlying memory. To work with the data, it implements a member method `at<Dims...>(idxs...)` that is used to index the data pointer with respect to the enveloping structure layout. This method is a wrapper for the library function `get_at`. Without using a `bag`, the indexing might look like this:

```
auto s = array<'j', 8, array<'i', 4, scalar<float>>>();
float* ptr = allocate_memory_bytes(s.size());
float x = s | get_at<'i', 'j'>(ptr, 2, 3);
```

The `bag` binds the layout together with data, systematizing the computation on the last line as follows:

```
auto b = bag(s, ptr);
float x = b.at<'i', 'j'>(2, 3);
```

Furthermore, to manage an explicitly bound external pointer, `bag` can also allocate the underlying memory automatically if no pointer is given (i.e., it also carries the semantics of a smart pointer). Technically, `bag` can belong to either one of two semantic groups according to the way it acquires data:

- *Owning semantics.* The bag is constructed only with a structure to envelop. The data pointer of exact length is automatically allocated using standard memory management (e.g., by `unique_ptr`), and the length is determined by calling `size()` on the wrapped structure.
- *Borrowing semantics.* The bag is constructed with both structure and data pointer. In this case, the deallocation, as well as ensuring the proper data-block length, has to be enforced by the caller.

5 Related Work

A significant group of works that touch the problem of memory layouts are parallel programming languages such as X10 [5], Chapel [4] or Legion [2]. Apart from providing syntax for simple parallel code expression, these languages allow for data decomposition into regions that can be mapped within the same memory space or more complex non-uniform memory spaces. Hence, the memory layout expression addressed by these works is only researched to the point of high-level data distribution among processing elements.

Application-specific library generators, or *active libraries*, also utilize memory layouts. The most known representatives are ATLAS [19], SPIRAL [15]

and FFTW [9] specializing in linear algebra, signal processing, and Fast Fourier Transform, respectively. They are trying to mitigate portability issues of manually optimized programs by selecting the best interprocedural optimizations for the hosting system using autotuning. Usually, these optimization strategies include some form of memory layout selection. It is important to note that active libraries target different stages in programming than Noarr; rather than performing the layout selection from the hardcoded set of layouts, Noarr provides means to *implement* such layout selections in a more extensible and object-oriented way.

The most related works we found are Kokkos [16], and GridTools [1]. These libraries allow the coupling of arbitrary data structures with memory layouts which can be either selected from a set of predefined layouts or programmatically customized.

GridTools specialize in block-structured grid applications such as combustion, seismic, and weather simulations, working with generalized stencil-like patterns. The library defines a storage infrastructure component that controls the layout, alignment, and padding of stored data fields. A layout is specified in code at compile time by selecting one of the predefined target backends, each well suited for a specific use case, such as vector instructions or GPU kernels. The library can be extended with new programmer-specified backends, but the layout can be altered only by permuting dimension order in a regular n-dimensional array.

An interesting approach is taken in the Kokkos library, which specifies the `View` class that couples the definition of data memory space, allocation, and layout altogether using C++ policy classes, yielding an object of similar functionality as our `bag`. The memory resource and allocation mechanism are abstracted and defined by the template argument. Kokkos provides multiple memory spaces such as `HostSpace`, `CudaSpace`, `CudaHostPinnedSpace`, thus representing CPU and GPU physical memory and their combinations.

In Kokkos, the memory layout is either implicitly deduced from the memory space or explicitly specified as another template parameter. The library implements row and column-major layouts together with the layout with strides with custom sizes. Kokkos allows user-defined memory layouts by defining a new layout policy and implementing a function that defines a bijective mapping between index space and memory addresses. However, this mapping must be defined on a regular n-dimensional array, using a minimal API that fits the `View` class.

Language-wise, our approach is similar to (and inspired by) known concepts from functional programming. Materialized, first-class composable references to sub-structures uncoupled from data have been extensively studied as optics [8]. In particular, the internal structures that implement the selection of array slices at certain indexes are similar to the concept of indexed lenses—kind of references that transparently provide information about the current index in a complicated structure, as summarized by Clarke et al. [6] In the future, it might be interesting to examine whether more advanced optics may be modeled in C++ for array

accesses, e.g., expressing repeated data accesses similarly to lens-based traversals or reconstructing the user-facing indexes from known offsets using isomorphisms.

6 Conclusion

We have presented a new high-performance approach for managing the complexity of offset computation in array-like data structures in modern C++. We introduced first-class layout structures that can be used to describe complex array layouts and run the required offset computations. The implementation is based on C++ template metaprogramming, exposing a rich interface for manipulating the structures with index mnemonics while enabling many compiler optimizations by properly separating static compile-time parameters and known constants from dynamic data.

The technique promotes complete decoupling of array indexing from memory allocation, which makes it applicable for many scenarios, including direct processing of memory-mapped files or re-using the same data structure layout in various memory spaces (e.g., offloading computations to GPUs). We showed that the layout structures, combined with the C++ templating system, make it easier to create layout-agnostic algorithms and functions, leading to a simpler selection of optimal layouts for a given hardware platform and problem configuration. Additionally, the utilization of layout structures makes it easier to create semi-automated layout transform routines, which can improve the performance of many algorithms.

We have implemented the proposed ideas in Noarr, a prototype library demonstrating the viability of the approach. We demonstrated the benefits in several examples and experiments; most importantly, we showcased the ability to write shorter program source code that promotes easier experimentation and compilation into faster solutions. The library is publicly available as an open-source portable to all mainstream compilers, including CUDA `nvcc`, and may be readily used in designing new libraries that consider performance a priority. We expect that the approach will simplify the research focusing on optimizations and automatic tuning of the performance of complex parallel algorithms.

Acknowledgements. This work was supported by Charles University institutional funding SVV 260451.

A Experimental Methodology

The main objective of the benchmarking was to measure the speedups achieved by different layout combinations to support the claims mentioned in the work[6]. A more complex performance evaluation is beyond the scope of this paper and is planned in future work.

[6] More details and the data are in the replication package https://github.com/asmelko/ica3pp22-artifact.

A.1 GPU Benchmarking Setup

In the results, we present mainly the kernel execution times measured by the high-precision system clock, which is available on all platforms. The relative standard deviations in 20 collected measurements of each result were less than 5% of the mean value in all cases, so we report only the mean values.

Due to the page limit, the presented results were limited to matrices of sizes (1008×1008) and $(10,080 \times 10,080)$. However, more extensive testing on other problem instances, including a broader range of matrix sizes and non-square matrices, exhibited similar results.

The results were collected on the following platforms:

- NVIDIA Tesla V100 SXM2 (Volta, CC 7.0, 1.3 GHz), Rocky Linux 8
- NVIDIA GeForce RTX 2060 (Turing, CC 7.6, 1.7 GHz), Windows 10
- NVIDIA GeForce RTX 3070 laptop (Ampere, CC 8.6, 1.6 GHz), Windows 11

All platforms used CUDA toolkit 11.6 with an up-to-date driver. These devices represent three of the most recent Nvidia architectures and three typical hardware platforms (server, desktop PC, and laptop). Hence, we claim that the measurements sufficiently represent contemporary CUDA-enabled GPUs.

A.2 CPU Benchmarking Setup

We ran the kernel in 100 iterations for the stencil benchmark, plotted the local regression outlining the mean value, and distinguished the outliers. The measurements were conducted using the following CPUs:

- AMD Ryzen 5 5600X (hi-end desktop CPU, 3.70 GHz), Windows 10
- Intel Core i7-10870H (laptop CPU, 2.20 GHz), Windows 11
- Intel Xeon Gold 5218 (server CPU, 2.3 GHz), Rocky Linux 8.

Due to the fact that some compilers may optimize `constexpr` expressions better than others, we compiled the benchmark using `clang++` v12 and `g++` v11 compilers with `-O3` flag. We also compiled the stencil benchmark using the MSVC C++ compiler, but the results showed that it could not sufficiently optimize Noarr code in the current version; hence, MSVC results are not included.

All benchmarking datasets were synthetic, with data sampled randomly from the same uniform distribution. We consider synthetic validation sufficient since the performance of the benchmarked algorithms is not data-dependent.

References

1. Afanasyev, A., et al.: GridTools: a framework for portable weather and climate applications. SoftwareX **15**, 100707 (2021). https://doi.org/10.1016/j.softx.2021.100707. https://www.sciencedirect.com/science/article/pii/S2352711021000522
2. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 1–11. IEEE (2012)

3. Bethel, E.W., Camp, D., Donofrio, D., Howison, M.: Improving performance of structured-memory, data-intensive applications on multi-core platforms via a space-filling curve memory layout. In: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, pp. 565–574. IEEE (2015)
4. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl. **21**(3), 291–312 (2007)
5. Charles, P., et al.: X10: an object-oriented approach to non-uniform cluster computing. ACM SIGPLAN Not. **40**(10), 519–538 (2005)
6. Clarke, B., et al.: Profunctor optics, a categorical update. arXiv preprint arXiv:2001.07488 (2020)
7. Clauss, P., Meister, B.: Automatic memory layout transformations to optimize spatial locality in parameterized loop nests. ACM SIGARCH Comput. Archit. News **28**(1), 11–19 (2000)
8. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. (TOPLAS) **29**(3), 17-es (2007)
9. Frigo, M., Johnson, S.G.: FFTW: an adaptive software architecture for the FFT. In: Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 1998 (Cat. No. 98CH36181), vol. 3, pp. 1381–1384. IEEE (1998)
10. Hawick, K.A., Playne, D.P.: Hypercubic storage layout and transforms in arbitrary dimensions using GPUs and CUDA. Concurr. Comput. Practice Exp. **23**(10), 1027–1050 (2011)
11. Heinecke, A., Bader, M.: Parallel matrix multiplication based on space-filling curves on shared memory multicore platforms. In: Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem, pp. 385–392 (2008)
12. Kruliš, M., Kratochvíl, M.: Detailed analysis and optimization of CUDA k-means algorithm. In: 49th International Conference on Parallel Processing-ICPP, pp. 1–11 (2020)
13. NVIDIA: CUDA C best practices guide (2013)
14. Panda, P.R., Semeria, L., De Micheli, G.: Cache-efficient memory layout of aggregate data structures. In: Proceedings of the 14th International Symposium on Systems Synthesis, pp. 101–106 (2001)
15. Püschel, M., et al.: Spiral: a generator for platform-adapted libraries of signal processing algorithms. Int. J. High Perform. Comput. Appl. **18**(1), 21–45 (2004)
16. Trott, C.R., et al.: Kokkos 3: programming model extensions for the exascale era. IEEE Trans. Parallel Distrib. Syst. **33**(4), 805–817 (2022). https://doi.org/10.1109/TPDS.2021.3097283
17. Weber, N., Goesele, M.: MATOG: array layout auto-tuning for CUDA. ACM Trans. Archit. Code Optim. (TACO) **14**(3), 1–26 (2017)
18. Weidendorfer, J., Ott, M., Klug, T., Trinitis, C.: Latencies of conflicting writes on contemporary multicore architectures. In: Malyshkin, V. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 318–327. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73940-1_33
19. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, SC 1998, p. 38. IEEE (1998)

Contribution 6
Noarr Traversors

Published as Jiří Klepl et al. "Pure C++ Approach to Optimized Parallel Traversal of Regular Data Structures". In: *Proceedings of the 15th International Workshop on Programming Models and Applications for Multicores and Manycores*. 2024, pp. 42–51

Pure C++ Approach to Optimized Parallel Traversal of Regular Data Structures

Jiří Klepl
Adam Šmelko
Lukáš Rozsypal
Martin Kruliš
klepl@d3s.mff.cuni.cz
smelko@d3s.mff.cuni.cz
krulis@d3s.mff.cuni.cz
Department of Distributed and Dependable Systems, Charles University
Prague, Czechia

Abstract

Many computational problems consider memory throughput a performance bottleneck. The problem becomes even more pronounced in the case of parallel platforms, where the ratio between computing elements and memory bandwidth shifts towards computing. Software needs to be attuned to hardware features like cache architectures or memory banks to reach a decent level of performance efficiency. This can be achieved by selecting the right memory layouts for data structures or changing the order of data structure traversal. In this work, we present an abstraction for traversing a set of regular data structures (e.g., multidimensional arrays) that allows the design of traversal-agnostic algorithms. Such algorithms can be adjusted for particular memory layouts of the data structures, semi-automated parallelization, or autotuning without altering their internal code. The proposed solution was implemented as an extension of the Noarr library that simplifies a layout-agnostic design of regular data structures. It is implemented entirely using C++ template meta-programming without any nonstandard dependencies, so it is fully compatible with existing compilers, including CUDA NVCC. We evaluate the performance and expressiveness of our approach on the Polybench-C benchmarks.

CCS Concepts: • **Computing methodologies** → **Parallel programming languages**; **Parallel algorithms**; • **Software and its engineering** → *Object oriented development*; *Compilers.*

PMAM '24, March 3, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0599-1/24/03
https://doi.org/10.1145/3649169.3649247

Keywords: Iteration order, Traverser, Memory optimizations, Parallel programming, Layout agnostic

ACM Reference Format:
Jiří Klepl, Adam Šmelko, Lukáš Rozsypal, and Martin Kruliš. 2024. Pure C++ Approach to Optimized Parallel Traversal of Regular Data Structures. In *The 15th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '24), March 3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3649169.3649247

1 Introduction

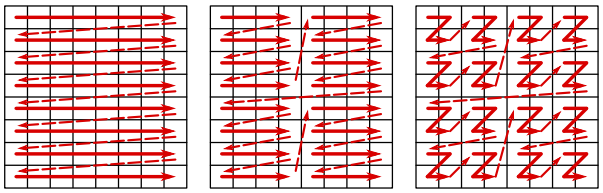
Memory operations are a cause of bottleneck in many situations. Contemporary CPUs dedicate a significant part of their circuits (such as multi-level caches or prefetching units) to mitigate this problem. In parallel processing, the situation becomes even more complicated as some resources are shared by the cores (like L3 cache, memory controllers, or memory buses), and the memory transactions need to be kept coherent (by MESI protocol, for instance). GPUs introduce another level of complexity caused by the lockstep execution model where multiple threads perform the exact instruction in the same cycle (so the memory transactions need to be planned across multiple cores) and by introducing special memory types like shared memory (with concurrently accessible banks).

The performance of many programs is often heavily affected by how they access data in the memory. If the data dependencies permit, the operations accessing the memory can be (re)arranged to take advantage of caching, prefetching, coalesced loads, parallel memory banks, or concurrent utilization of memory controllers without affecting the semantics (i.e., the results) of the algorithm. Even when the (re)arrangement does not change the number of operations, it may reduce the execution time if the latencies of the data transfers decrease. Unfortunately, the optimal arrangements are often system-specific and rather difficult to find.

This paper focuses mainly on regular data structures with multidimensional indexing (such as matrices, tensors, or grids). Such a data structure defines its indexing space (i.e.,

dimensions) and a mapping from the indexing space into the (linear) memory addressing space. The actual memory access pattern is then affected by the layout mapping and how its indexing space is traversed.

Let us illustrate the problem on a common matrix. It defines the index space (i, j), where the dimensions run from 1 to H (height) and W (width) respectively. A matrix can be stored in many ways (Figure 1). Perhaps the most common is the *row-major* order, where linear offsets are computed as $i \cdot W + j$. If the matrix is traversed by two nested for-loops (over i and j), the memory will be accessed sequentially, which often performs optimally on contemporary CPUs. If we swap the loops (j will become the outer loop), the subsequent memory operations will be W elements apart, which disrupts the prefetching and may increase cache misses.



(a) row-major **(b)** tiled **(c)** z-curve

Figure 1. Examples of common matrix layouts

Transforming the layout of a data structure or the order of its traversal may have a profound effect on the performance [8]. Although the compilers attempt to optimize these operations (e.g., by application of polyhedral optimizer to reorder nested loops), these automated efforts do not always meet with success since the compilation is bound with strict assumptions about data dependencies and alignment, the transformation search space is vast, and it is often difficult to predict the impact a transformation has on performance. Designing such transformations manually may prove difficult, tiresome, and even error-prone, especially in the domain of parallel applications. Therefore, it might be beneficial to provide the programmer with code constructs that would allow for explicit yet simple and flexible ways of expressing the desired transformations of traversal order.

In this work, we present an abstraction that facilitates a flexible specification of traversals of regular data structures. Our proposed implementation is an extension of C++ library *Noarr*[1], which provides first-class structures for defining memory layouts [15]. Our extension (*Noarr Traversers*) uses the same design philosophy (templated first-class transformation structures) for semi-automated traversal (over the indexing space) and provides basic transformation elements such as loop interchange, strip-mining, tiling, or z-curve. The proposed solution has the following benefits over contemporary libraries and tools that aim at the same problem:

1. *Standard compilers support*: The abstraction is defined in standard C++ and does not require any compiler extensions or domain-specific language (DSL) preprocessing, which is usually the case with annotation-based and DSL-based frameworks such as Loopy [12] or Halide [14].
2. *First class transformations*: A transformation is assembled from prepared templated classes and instantiated as a first-class object that is then applied in an algorithm written using traversal-agnostic loop constructs. This promotes code reusability (multiple versions of an algorithm are produced by applying different transformations), and it also allows constant parameters to be embedded in the type, moving some of the computation into compile time.
3. *Custom transformations*: The user has the expressive power of an imperative language (C++) to define custom transformations, not being limited by the syntax of annotation-based frameworks or restricted DSLs.
4. *Suitable for parallelism*: The proposed framework is designed to be easily utilized on various parallel platforms and libraries, namely multicore CPUs (TBB) and manycore GPUs (CUDA).

The aforementioned benefits should simplify coding when dealing with manual optimizations. More importantly, we aim to create an ecosystem where this abstraction can be used for semi-automated optimizations using autotuning or machine learning models. In such systems, designing the code in a traversal-agnostic way (or the data structures in a layout-agnostic way) simplifies the injection of the layouts or traversal patterns by the external optimizer. This continuation of the work will be accompanied by a careful assessment of the increase in compilation time limiting the exploration of the possible transformations.

Let us emphasize that the aforementioned benefits define the intended group of users for our tool. Other approaches may be better (lead to faster implementations or require less code to write) in cases where some of the benefits are considered irrelevant. For instance, using a specific DSL may be easier in simpler cases (Halide [14]) at the cost of universality and the necessity for more compilation steps.

The paper is organized as follows. Section 2 explains Noarr and introduces the running examples. The proposed abstraction is explained in Section 3, and Section 4 describes its utilization for parallel programming (TBB and CUDA). Section 5 presents the evaluation results. The related work is summarized in Section 6 and Section 7 concludes the paper.

2 Background

The problem of memory layouts and traversal order of data structures can be tackled using various approaches (besides the automated optimizations performed by the compiler):

[1] https://github.com/ParaCoToUl/noarr-structures

- *Native approach* uses only native constructs of the selected language. In C++, for instance, class policies can be used for selecting data structure layouts and iterators for data structure traversal.
- *Annotations* may be introduced into the language to hint to the compiler how the data structures (e.g., arrays) or loops may be transformed. This approach usually builds on native compiler optimizations (e.g., to guide polyhedral optimizer [12]), but it also requires specialized compilers or compiler plugins.
- *Domain specific language* (DSL) may describe either a data structure or the computation kernel in an abstract form. If the DSL is restricted and the target problem is simple enough, its compiler can extract an optimal execution plan for the kernel, not only optimizing memory operations but possibly handling the scheduling of parallel execution as well [14].

We investigate the native approach; however, our objective is to step beyond the traditional design patterns and software engineering practices. We aim at exploiting the possibilities of C++ language to its limits using templates, functional-like assembly of data types, and static (compile-time) metaprogramming.

2.1 Noarr structures

We base our solution on the Noarr library [15], which provides an abstraction for creating data structure layouts. The key idea is that the layout is represented by a first-class structure. The type of a Noarr structure is assembled from predefined templated base types like arrays, vectors, or tuples. The following example shows two representations of a matrix — row-wise (`rw`) and col-wise (`cw`). Let us emphasize the arguments `'i'` and `'j'` which identify the dimensions.

```
auto rw = scalar<float>() ^ vector<'j'>() ^ vector<'i'>();
auto cw = scalar<float>() ^ vector<'i'>() ^ vector<'j'>();
```

The two structures define the abstract layout of a matrix. The structures are immutable, and each can be used as a basis for creating various structures with fixed sizes, for example:

```
size_t size = ...;
auto matrix_struct = rw ^ noarr::set_length<'i', 'j'>(size, size);
```

In this case, the matrix size is set at runtime, and so the size is stored in the object; however, using the same syntax can embed the sizes in the type so they are computed at compile-time (for example, if `size` was `noarr::lit<42>`).

Another important principle of Noarr is decoupling the layouts from memory management. The structures used in the previous examples have no binding to memory. They represent an indexing abstraction for computing linear offsets, which can be used in internal and external data structures alike, or it can be used with any base pointer to dereference memory values:

```
size_t offset = matrix_struct | noarr::offset<'i', 'j'>(i, j);
float &ref = matrix_struct | noarr::get_at<'i', 'j'>(ptr, i, j);
```

Since most data structures reside in the main memory, Noarr offers a wrapper called *bag*, which binds the Noarr structure with a pointer. If we do not specify a memory location for the data represented by the Noarr structure, the bag automatically allocates the memory on the heap. However, the user can also specify a memory location (e.g., a memory-mapped file or a shared memory in the CUDA kernel), and the bag will use that instead.

```
auto matrix = noarr::make_bag(matrix_struct);
float &ref = matrix.template at<'i', 'j'>(i, j);
```

The current implementation of the bag does not employ any more complex memory management operations like host-device memory transfers or memory mapping. Such operations need to be controlled by the user of Noarr. The bag merely specifies indexing semantics on top of a pointer. However, extending this abstraction to a more complex behavior in the future is technically possible.

2.2 Running examples

In this section, we detail two running examples that we will use to demonstrate the syntax and the benefits of the proposed abstraction in the later sections.

2.2.1 Matrix multiplication. It presents one of the most profound problems with many applications. Being a well-studied problem, we can draw on the known optimizations and express them using our abstractions. We rely on the naïve $O(N^3)$ algorithm, which computes elements of the output matrix as dot products. Having square matrices `A` and `B` (of the size N^2), the product matrix `C` may be computed as:

```
for (size_t i = 0; i < N; ++i) {
    for (size_t j = 0; j < N; ++j) {
        C[i][j] = 0;
        for (size_t k = 0; k < N; ++k) {
            C[i][j] += A[i][k] * B[k][j];
}   }   }
```

The individual elements of the output matrix can be computed independently (even concurrently), and the internal dot products are both associative and commutative, allowing more fine-grained optimizations. Typical optimizations are based on tiling, which requires splitting the outer two loops and may also enable efficient parallel processing [11].

2.2.2 Histogram. An approximation of the distribution of numeric data often used in data analysis and related fields (e.g., machine learning or similarity search). The objective is to assign data elements into predefined bins (categories) and count the number of elements in each bin. Having histogram `H` and a function that finds a bin for each element, the algorithm can be coded simply as:

```
for (size_t i = 0; i < N; ++i)
    H[findBin(data[i])] += 1;
```

The histogram algorithm is particularly interesting from the perspective of parallel computing [2]. When the input elements are processed concurrently, the histogram updates

must be synchronized (e.g., by atomic instructions). If the number of bins is low and the level of concurrency high (typically on a GPU), the histogram updates will become a bottleneck. In such cases, sophisticated methods of privatization (and subsequent merging of private copies) could be beneficial. Another perspective is that a histogram can be computed as a bin-wise parallel reduction (with per-bin data filtering). These issues will help us to demonstrate the capabilities of the proposed abstraction.

3 Proposed Abstraction

We propose an abstraction for flexible traversal of regular data structures. This abstraction is implemented as an extension (named *Noarr Traversers*) of the C++ library Noarr. The extension applies the fundamental Noarr approach of specifying data layouts via a composition of elemental first-class objects (called *proto-structures* in Noarr) to the transformations of traversal orders.

A *traverser* is a first-class object that represents an index space and its corresponding traversal order. It is constructed from one or multiple Noarr structures to be traversed together. The traverser constructs the base index space from the combination (unification) of dimensions of the provided structures. The user can then provide a callable object (usually a lambda expression) that specifies the action performed on elements indexed by each point of the index space. For a single structure, this corresponds to the for-each algorithm. For two structures presenting the same set of dimensions (but not necessarily the same layout), the traverser can be used, for example, to copy the values from one structure to the other, which can implement transposition — this generalizes to other common algorithms such as reduction if we transform the dimensions of the input structures accordingly.

To alter the traversal order of the index space, the traverser can be transformed by applying a transformation structure, producing a new traverser. The transformation structure is assembled from elemental first-class proto-structures in a similar way to Noarr structures. The proto-structures defined for this purpose represent basic loop transformations such as loop interchange, strip-mining, tiling, z-curve, or more general transformations such as introducing new loops, binding some iteration dimensions to specific indices, or restricting their spans. The transformation structure can be defined separately and reused for different traversers.

Transforming the traversers by applying a separate object from the outside enables a simple way to design traversal-agnostic algorithms. We can then create multiple versions of the same computation by applying different transformations to the same traverser.

A traverser can also be used as an argument to a parallel executor, which then performs the traversal in parallel (we have implemented one based on TBB and one for CUDA, as examples). The parallelization is guided by one or multiple dimensions of the traverser, and each started thread is provided with an *inner traverser* representing the traversal of its corresponding traversal section that is usually constructed via binding some dimensions to specific values.

3.1 Introducing syntax for traversers

A traverser is constructed and executed in three steps that also denote the three key principles:

1. The constructor of the traverser is given one or more Noarr structures and deduces the *base index space* from them by unifying their dimensions.
2. A transformation is applied to the base index space via the `.order(transformation)` method. It changes the traversal order of the individual points of the index space. This step is optional and possibly reoccurring (composing the provided transformations into one).
3. The `.for_each(action)` method is called, where the actual body of the traverser (usually a lambda function) is injected. This provides a uniform interface that can be used for sequential iteration and parallel processing.

When constructed using a single structure, the traverser iterates through the cartesian product of the dimensions of that structure and calls the provided lambda function (body) with a tuple-like *state* object. The state object represents a point in the index space that can be used to access the corresponding element of the traversed structure. The following example performs an element-wise initialization of structure `c` (like a traditional for-each algorithm):

```
noarr::traverser(c).for_each([=](auto state) { c[state] = 0; });
```

The traverser can properly combine the indexing space from multiple Noarr structures by creating a cartesian product of different dimensions while unifying matching dimensions based on their names (template identifiers). If we name indices of three matrices $\mathsf{a}(i,k)$, $\mathsf{b}(k,j)$, and $\mathsf{c}(i,j)$ the matrix multiplication can be written simply as:

```
noarr::traverser(a, b, c).for_each([=](auto state)
    { c[state] += a[state] * b[state]; });
```

The traverser extracts dimensions i, k, k, j, i, j, which (after unification) yields the indexing space to be the cartesian product of (i,k,j). In other words, the index space corresponds to the three nested loops of the naïve matrix multiplication.

The traverser and its index space can be transformed using the `order()` method, which takes a transformation structure as an argument. In the case of matrix multiplication, the most common transformation would be to perform tiling — i.e., splitting each of the indices into an index of a block (of fixed size) and a local index within the block. An example of such transformation is presented in the following.

```
auto blocks = noarr::strip_mine<'i', 'I', 'i'>(noarr::lit<16>)
    ^ noarr::strip_mine<'k', 'K', 'k'>(noarr::lit<16>)
    ^ noarr::strip_mine<'j', 'J', 'j'>(noarr::lit<16>);

noarr::traverser(a, b, c).order(blocks).for_each([=](auto state)
    { c[state] += a[state] * b[state]; });
```

Note that the transformation structures can be declared separately so they can be reused for different traversers (and vice-versa). The `strip_mine` template performs *tiling* where the first index denotes the dimension to be tiled, and the second two denote the newly created dimensions (existing dimensions are replaced). The tiling is followed by *hoisting*, which moves the first of the two created dimensions into the outermost traversal loop. The `noarr::lit<16>` ensures the constant tile size is embedded into the type.

In some situations, it is beneficial to iterate over whole sections of the index space instead of single values. A typical example of that is accumulating a portion of the dot product corresponding to a given block in a local variable (a register) to reduce the number of memory operations. In such cases, we replace `for_each` call with templated `for_dims`, which takes a list of dimensions that represent the sections to be traversed. It creates an instance of an *inner traverser* corresponding to the current index space section. The inner traverser offers the same traverser interface, so it can be used for an internal traversal over the given section without changing the body of the traversal.

```
noarr::traverser(a, b, c)
  .order(blocks)
  .template for_dims<'I', 'J', 'K', 'j', 'i'>(
    [=](auto inner_trav) {
      auto res = c[inner_trav.state()];
      inner_trav.for_each([=, &res](auto state) {
        res += a[state] * b[state];
      });
      c[inner_trav.state()] = res;
  });
```

There are many transformations already implemented in Noarr. That includes renaming and reordering the indices, restricting iteration spans and slicing, fixing indices in particular dimensions, and some more complex operations designed for parallel processing. Details can be found in our replication package[2].

4 Parallel Execution

Besides the benefits granted by the iteration order agnosticism of traversers, the abstraction can easily be extended to parallel processing. A parallel *for-each* example would be trivial, so we start with parallel reduction.

```
auto in = make_bag(scalar<char>() ^ sized_vector<'i'>(size), i);
auto out = make_bag(scalar<size_t>() ^ array<'v', 256>(), o);
noarr::traverser(in).for_each([=](auto state) {
    out[noarr::idx<'v'>(in[state])] += 1;
});
```

The demonstration is based on the histogram running example (Section 2.2). The sequential implementation (presented above) comprises a simple for-loop. The `in` variable is a bag (a wrapper that combines Noarr structure with memory pointer `i`) holding the input (vector of `char`) and `out` is a bag holding the histogram (256 bins stored in `o`).

We have decided to design the parallel executors as external tools that take a traverser as an argument instead of extending the traverser interface. This approach is more modular and can be easily extended by implementing new parallel executors using various libraries (C++ standard library, TBB [13], or OpenMP [7]). As a proof of concept, we present a TBB implementation of the parallel reduce algorithm wrapper for Noarr traversers.

```
noarr::tbb_reduce_bag(
    noarr::traverser(in),
    [](auto out_state, auto &out_left) {
        out_left[out_state] = 0;
    },
    [in](auto in_state, auto &out_left) {
        out_left[noarr::idx<'v'>(in[state])] += 1;
    },
    [](auto out_state, auto &out_left, const auto &out_right) {
        out_left[out_state] += out_right[out_state];
    },
    out);
```

The `tbb_reduce_bag` algorithm template takes five arguments. Besides the traverser and the output bag, there are three lambdas — the first initializes the output structure to zero, the second performs the element-wise reduction, and the third performs the merging of privatized copies of the output structure (histogram).

The reduction is performed automatically over the whole space defined by the traverser, but only the first dimension is processed concurrently. The user can explicitly change which dimension is the first by applying `.order` to the traverser, thus affecting the parallel decomposition.

Privatization of the output structure is performed transparently to prevent data collisions. If the `out` structure is parametrized by the iterated dimension, then the different workers access different places in the memory, and no privatization is necessary. Otherwise, the algorithm creates a local copy of the `out` structure for each worker thread as needed (managed by `tbb::combinable`), allocating appropriate memory when the given copy is used for the first time. The copies are merged at the end using the third lambda.

4.1 Extension to GPU (CUDA traverser)

One of the key advantages of the proposed abstraction is that it aims at maximal compatibility with standard C++ compilers. This simplifies and expedites its application within other parallel environments like CUDA, which employs its custom compiler that adds some extensions but remains compatible with C++ language. We present an adaptor that allows applying traversers for kernel execution and one particular construct that becomes especially useful when privatizing data structures in shared memory.

CUDA framework is based on the data-parallel paradigm and uses thread abstraction to achieve parallelism. CUDA threads are spawned collectively (forming a *grid*) executing a single piece of code (*kernel*). Each thread is given index structures (`threadIdx`, `blockIdx`), which identify a data element to be processed by the thread. Additionally, threads are grouped into *thread blocks* so they can cooperate more

[2] https://github.com/jiriklepl/PMAM2024-artifact

closely (e.g., via *shared memory* or using faster synchronization primitives). The indexing structures (for threads and blocks) can encompass up to three dimensions, so the model is more convenient for programmers when dealing with multidimensional data (like matrices or 3D grids).

From the perspective of traversers, the CUDA grid is mapped to selected loop dimensions. The original traverser can be transformed to achieve the desired mapping — i.e., which parts of the traversal are executed (possibly) concurrently and which are handled inside a CUDA thread. The following code represents a kernel that computes the histogram (stored in global memory) using atomic updates (a typical implementation) and where each thread computes multiple input values. The aggregation of work per thread is one of the common optimizations. In this case, it could produce more coalesced loads from global memory and prepare grounds for more elaborate optimizations like shared memory privatization, which we discuss further in the paper.

```
template<class InTrav, class In, class Out>
__global__ void histogram(InTrav in_trav, In in, Out out) {
    in_trav.for_each([=](auto state) {
        auto value = in[state];
        atomicAdd(&out[noarr::idx<'v'>(value)], 1);
    });
}
```

`in_trav` is an inner traverser created from the traverser of the input data in the kernel invocation (see below), and it covers the data traversed by a single thread. The invocation is handled as follows.

```
auto in_blk_struct = in_struct
    ^ noarr::into_blocks<'i', 'B', 't'>(BLOCK_SIZE)
    ^ noarr::into_blocks<'B', 'b', 'x'>(ELEMS_PER_THREAD);
auto in = noarr::make_bag(in_blk_struct, in_ptr);
auto out = noarr::make_bag(out_struct, out_ptr);

auto ct = noarr::cuda_threads<'b', 't'>(noarr::traverser(in));
histogram<<<ct.grid_dim(), ct.block_dim()>>>(ct.inner(), in, out);
```

The essential part of the mechanism is hidden in the function `cuda_threads` that automatically associates the dimensions of the traverser with the dimensions of the CUDA grid — in this case, letting the `b` be the index of the block and `t` the index of the thread within the block. The resulting *cuda traverser* is then used to provide kernel invocation parameters by `grid_dim()` and `block_dim()` calls and infer the inner traverser that is passed as an argument of the kernel. The inner traverser binds its `b` and `t` dimensions to the `blockIdx` and `threadIdx` structures respectively, and allows (in-thread) iteration over the remaining dimension `x`.

Let us emphasize that the execution, as well as internal behavior (how many items are processed by a thread), are both governed by the traverser. That permits a certain level of agnosticism in the parallelization of algorithms. Furthermore, the composable nature of traversers makes it possible to separate the blocking operations required for CUDA execution to be prepared in a separate structure applied by `order()` method. Furthermore, since the kernel invocation is a common operation, Noarr also provides a method `simple_run()`, which can be used instead as a shortcut.

4.2 Shared memory privatization

Massively parallel systems are particularly susceptible to intensive data synchronization. In the `histogram` kernel presented in the previous section, the many simultaneous atomic updates cause a bottleneck. Even if the updates are distributed evenly, collisions are unavoidable since the histogram has much fewer bins than the GPU has cores.

A typical solution to this problem is data structure privatization — i.e., creating multiple copies of the histogram so each thread (or a small group of threads) has a separate copy. In this case, the optimal solution is to create a copy for each warp lane (32 copies per thread block) and place it in the shared memory. This way, threads running in lockstep have no collisions among themselves. The result aggregation in the shared memory significantly decreases the number of global memory accesses. Then, the individual copies need to be merged into the final copy in the global memory before a thread block concludes its execution.

The shared memory has a specific hardware design — it is divided into 32 independent memory banks (consecutive 32-bit words are placed in banks in a round-robin fashion), so each thread in the warp can access a different bank. Concurrent operations accessing one bank are serialized (except for special cases like data broadcast), which delays an entire warp. Histogram stored in a contiguous block in the shared memory would span over all banks, so concurrent updates would still cause bank conflicts (and thread serialization) even if the structure is privatized. The solution is to place each histogram copy into a separate bank, which requires a rather specific stridden layout pattern.

We introduce `noarr::cuda_striped<N>`, a helper structure tailored particularly for shared memory. The parameter N denotes the number of copies distributed across the banks. The optimum is $N = 32$ (i.e., one copy per bank); however, picking a lower N may be necessary if 32 copies would not fit in the memory. The kernel could be optimized using a striped structure `shm_s`, as follows. (For the sake of brevity, we omit initialization, reduction, and the necessary barriers.)

```
template<class InT, class I, class ShmS, class O>
__global__ void histogram(InT in_trav, I in, ShmS shm_s, O out) {
    extern __shared__ char shm_ptr[];
    auto shm_bag = make_bag(shm_s, shm_ptr);
    // initialize shared memory (zero the bins)
    in_trav.for_each([=](auto state) {
      auto val = in[state];
      atomicAdd(&shm_bag[noarr::idx<'v'>(val)],1);
    });
    // reduce shm copies and write the histogram in global memory
}
```

Note that the `atomicAdd` merely uses the bag allocated in the shared memory, and the `shm_s` structure transparently ensures the appropriate privatized copy is accessed (based on the `threadIdx` value). The construction of `shm_s` structure is performed externally (as well as the shared memory allocation) in our example, so the kernel is more generic, and the shared memory utilization can be subjected to external tuning; however, if required, it can be constructed internally.

```
// 'in' and 'out' match the previous example
auto ct = noarr::cuda_threads<'b', 't'>(noarr::traverser(in));
auto shm_s = out_struct ^ noarr::cuda_striped<NUM_COPIES>();
histogram<<<ct.grid_dim(), ct.block_dim(),
    shm_s | noarr::get_size()>>>
    (ct.inner(), in, shm_s, out);
```

The shared memory needs to be initialized when each thread block starts — in this particular case, all histogram copies need to have their bin counters zeroed. The most efficient way is for all threads (of a block) to cooperate on initialization evenly. For this purpose, we use `noarr::cuda_step`, which automatically distributes the work among the available threads. The `cuda_step` object is constructed using the rank of the current thread and the number of threads cooperating on the stripe provided by `current_stripe_cg`.

```
auto subset = noarr::cuda_step(shm_s.current_stripe_cg());
noarr::traverser(shm_bag).order(subset).for_each(
    [=](auto state) { shm_bag[state] = 0; });
```

A different access pattern is required at the end, where the histogram copies are merged. In this case, the threads cooperatively iterate over the histogram, processing the bins concurrently. Each bin is summed up across the copies and atomically added to the global structure. The `num_stripes` method returns the number of copies. The difficulty here is that we cannot access the shared memory bag directly since it would direct each thread to its corresponding copy, so the actual index (state) needs to be computed as follows.

```
noarr::traverser(out).order(noarr::cuda_step_block())
  .for_each([=](auto state) {
    size_t sum = 0;
    for (size_t i = 0; i < shm_s.num_stripes(); ++i) {
      sum +=
        shm_bag[state.template with<noarr::cuda_stripe_index>(i)];
    }
    atomicAdd(&out[state], sum);
  });
```

Granted, the code required to access all private copies from each thread is rather complex. However, this type of access is required only for the final reduction, and such an operation can be easily wrapped in a templated algorithm, so the regular user would not have to implement it explicitly.

5 Evaluation

The evaluation has two objectives: We would like to demonstrate that the proposed abstraction has no additional performance overhead, and we discuss its qualities from the perspective of the programmers using simple code metrics.

The most important results are presented in the remainder of this section. The complete set of experiments and results is available in the replication package.

5.1 Methodology and datasets

The presented experiments were measured on Intel Xeon Gold 6130 (CPU) and Tesla V100 PCIe 16GB (GPU) compiled with GCC 12.2 and NVCC 12.2. Each test comprised one warmup run and 10× subsequent measured runs on the `EXTRALARGE` dataset. The wall time of the tested kernel was measured by a high-resolution system clock. As expected, the variance of the measured times was very low (below 1%), so we present only the mean values.

We used *Polybench/C-4.2.1*[3] and *Polybench/GPU-1.0*[4] [9] benchmark suites for the performance evaluation. The Polybench/C suite (CPU kernels) contains a set of 30 algorithms commonly used in scientific high-performance computing, such as problems from linear algebra, stencils, or data mining. The Polybench/GPU suite contains a set of 21 algorithms mostly from the Polybench/C suite, with the addition of some algorithms that are more specific to GPU computing (such as 2DConvolution). For Polybench/GPU, we have implemented 5 algorithms as a representative subset for the evaluation.

5.1.1 Threats to validity.

The greatest concern is whether our Noarr implementation is comparable with the original Polybench code. To mitigate this threat to validity, we have imposed several rules that govern the transcription of Polybench kernels into their Noarr counterparts:

1. All data layouts are equivalent; each dimension of a data structure is represented by `noarr::vector`.
2. The loops from the baseline implementation are directly mapped to equivalent Noarr iterative constructs (such as methods `for_each` and `for_dims`).
3. Kernels are structurally equivalent, and their computation statements are in the same order and rewritten into an equivalent form.
4. Accesses into data structures are at the equivalent computation points.
5. The time measurements and device synchronizations (for GPU) take place at equivalent program points.

Rewriting the algorithms according to these requirements is not easily automatable and it takes an extensive programming effort. However, as a sanity check, we have included scripts that check whether the implementations produce the same result.

5.2 Performance results

The performance results are presented as a relative speedup of Noarr implementations over their corresponding plain C/C++ (or CUDA) counterparts. Speedups above 1× indicate that the Noarr implementation enabled additional compiler optimizations, whereas speedups below 1× indicate possible overhead or that Noarr prevented some optimizations.

Figure 2 summarizes the results of the entire Polybench in sequential execution. Most of the algorithms indicate that Noarr implementation has the same performance as plain C. There are four outliers where Noarr performed better and four where it performed worse than the baseline. Examining the compiled code indicates that the differences are caused by the compiler selecting a different optimization path.

[3] https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1
[4] https://github.com/sgrauerg/polybenchGpu

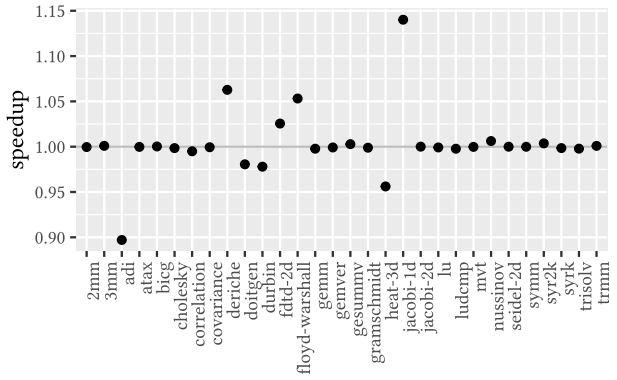


Figure 2. Comparing Noarr to plain C on Polybench/C-4.2.1

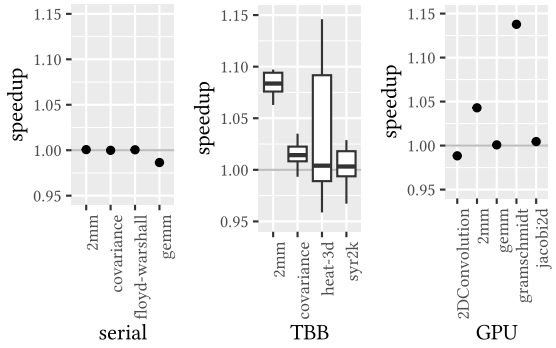


Figure 3. Comparing selected algorithms Noarr vs. plain C/C++/CUDA: tuned for performance (left), TBB parallelization (middle), GPU parallelization (right)

Figure 3 (left) presents the speedups of a selected subset of Polybench algorithms that were subjected to tuning (applying tiling and loop reordering). The middle graph presents the results of selected algorithms with their outermost loop in the critical segment parallelized using TBB. The GPU results (using CUDA traverser) are presented in the right graph of Figure 3. The results indicate that neither the additional traverser transformations applied in Noarr nor the parallelization extensions have any significant overhead over direct implementation in C, TBB, and CUDA, respectively. The parallel processing on a multi-socket CPU host is much more volatile, so we present the boxplots of all ten results instead of the mean value in the TBB graph.

5.3 Discussing code design aspects

Comparing the loop transformation approaches from the code design perspective is very challenging for many reasons. A user study might be the best way, but it is currently beyond our capabilities as it would require the cooperation of many users. For the basic insight, we provide a discussion comparing three typical approaches (annotations, DSL, and native C++ with the assistance of Noarr). Details about our selection of the compared technologies are in Section 6. We use the matrix multiplication running example optimized for memory transfers by blocking.

```
float A[I][K], B[K][J], C[I][J];

for (i = 0; i < I; i++)
    for (j = 0; j < J; j++)
        for (k = 0; k < K; k++)
Comp:       C[i][j] += A[i][k] * B[k][j];

affine(Comp, {[i,j,k]->[i,k,j]})
affine(Comp, {[i,j,k]->[i1,j1,k1,i2,j2,k2]: i1=[i/32] and i2=i%32
     and j1=[j/32] and j2=j%32 and k1=[k/32] and k2=k%32})
```

listing 1. Loopy (using `affine` compiler directives)

Listing 1 presents an implementation that relies on annotations. It keeps the code quite close to the original (plain C) implementation since the entire transformation is described by separate `affine` constructs. On the other hand, these constructs are quite complex to understand at first glance and limited to affine transformations only.

```
Halide::Buffer<float> A{I, K}, B{K, J}, C{I, J};

Halide::Func Comp{"Comp"};
Halide::Var i{"i"}, j{"j"};
Halide::RDom k{0, K};

Comp(i, j) = C(i, j); // Initial values
Comp(i, j) += A(i, k) * B(k, j); // Matrix multiplication

Halide::Var i2{"i_inner"}, j2{"j_inner"};
Halide::RVar k1{"k_outer"}, k2{"k_inner"};

Comp.update().reorder(i, k, j)
   .tile(i, j, i2, j2, 32, 32).split(k, k1, k2, 32);

Comp.realize(C);
```

listing 2. Halide (DSL using methods on function stages)

The Halide implementation (Listing 2) represents the DSL approach. Halide was designed for regular operations like matrix multiplication; thus, the realization is easy, albeit a little more verbose than Loopy and Noarr. On the other hand, with more complex data dependencies or irregular data traversals (for instance, the Gram-Schmidt algorithm from Polybench), Halide implementation gets quite cumbersome.

```
auto A = bag(scalar<float>() ^ array<'k', K>() ^ array<'i', I>());
auto B = bag(scalar<float>() ^ array<'j', J>() ^ array<'k', K>());
auto C = bag(scalar<float>() ^ array<'j', J>() ^ array<'i', I>());

auto my_order = into_blocks<'i', 'I', 'x'>(32) ^
                into_blocks<'j', 'J', 'y'>(32) ^
                into_blocks<'k', 'K', 'z'>(32) ^
                reorder<'I', 'K', 'J', 'x', 'z', 'y'>();

traverser(A, B, C).order(my_order).for_each([&](auto state) {
    C[state] += A[state] * B[state];
});
```

listing 3. Native C++ with Noarr traversers

Finally, Listing 3 presents our implementation in Noarr. The complexity is comparable both with Loopy and Halide, though the assembling of structures and traverser ordering

may seem a little unusual for mainstream C++ programmers since it uses functional programming patterns. The greatest benefit is that the type constructs for structures and orderings can be easily reused, which simplifies the design of similar data structures and the optimization of similar algorithms. Furthermore, this code can be compiled by any C++ standard-compliant compiler without extra preprocessing.

To assess the implementation overhead of Noarr compared to simple C code, we extracted the corresponding kernel codes delimited by the `scop` pragmas and formatted them using `clang-format`. We then compared them using coding metrics. On average, a Noarr implementation contains 11.28% more lines of code and 31.95% more individual code tokens than the baseline implementation. When compressing each kernel with gzip, the average Noarr implementation is 41.19% larger than the C baseline. This figure drops to 28.67% when comparing the gzipped tar archives containing all kernels. These results demonstrate that direct reimplementation using the proposed abstraction increases the size of the source code by approximately a third. However, the added code agnosticism makes the proposed approach superior when there is a need for at least two versions of the same algorithm (eliminating the need for code duplication) or when frequent modifications in the traversal order are required (handled by updating just the transformation structure).

6 Related Work

Optimization based on loop transformations has been addressed from various perspectives in vast research materials, namely in the fields of compilers, vectorization, autotuning, code generators, and optimizations of particular scientific computations. Contemporary compilers use sophisticated loop optimizers based on the polyhedral model, such as Graphite in GCC [16] or Polly in LLVM [10]. However, these optimizers are limited by the lack of information about the effects of the transformations on the optimized metric.

One of the first papers [6] that addressed the loop transformations from the perspective of optimizing memory operations is over 20 years old. It proposed using Ehrhart polynomials to compute how many times a single index reference is computed in a loop. Since then, several models based on static predictions have been created [10, 16]. The most recent innovations focus on elaborate multi-objective scheduling for loop transforms [4].

Autotuning methods have also addressed this problem by generating various variations of the tuned program and evaluating them either by sophisticated models or by measuring execution metrics such as execution time. Modern autotuning tools are often built on top of existing optimizers and employ methods from the machine-learning domain — for instance, Wu et al. [18] presented a tuning tool based on Polly [10] that employs Bayesian optimizations.

6.1 Domain Specific Languages

Many works address the issue of separating the specifics of memory access patterns and traversals from the algorithm itself by defining the algorithm via some DSL with a simplified model that facilitates applying various transformations. We have selected two representatives used in state-of-the-art production code. Our approach can be superficially related to theirs, with the fundamental distinction of their approach relying on a custom compilation pipeline and a runtime library, while Noarr is compiled by standard C++ compilers.

The Halide language [14] follows a decoupling approach similar to our combination of traversers and proto-structures. Halide primarily focuses on image processing, but their approach found use even for optimizing deep learning algorithms, as shown by the work of Apache TVM [5]. In their approach, the definition of an algorithm is followed by a *schedule* that represents various traversal transformations. The schedules roughly correspond to our idea of traversers and their transformations via proto-structures, but they lack any support for more complex or user-defined traversals (such as the z-curve).

6.2 Annotations

Another approach employed by various tools and compiler extensions uses code annotations that suggest the desired way of handling data structure layouts or loop transformations. Loopy [12] is a system for loop transformations designed as an extension to the LLVM compiler, which is perhaps the closest to our research since it relies on programmer-guided loop transformations. Building upon a polyhedral compilation library, it provides custom affine transformations and testing for the legality of loop transformation.

6.3 Native tools

The projects closest to our approach can be characterized by being built using abstractions provided by the C++ language itself and thus allowing for more seamless interaction with other C++ features, various intrinsics, or user-defined abstractions. This also avoids the necessity for custom development toolkits in favor of existing tools for C++ development, greatly reducing requirements on maintenance.

The C++ Standard Library already provides an abstraction for different traversal options via its ranges library. However, the library is not designed for parallelism and does not support multidimensional data layouts. The most common layouts of multidimensional arrays are expressible via the `mdspan` class template, but this abstraction lacks generality and does not provide a way of expressing traversals.

The NVIDIA Thrust library [3] provides routines for parallel code execution on both CPU and GPU. It is a template library based on the C++ Standard Library. While providing plenty of freedom in defining systems-agnostic concurrent traversals via functions like `thrust::for_each` or

`thrust::reduce`, their approach is based on an iterator design pattern restricted to 1D traversals. Furthermore, Thrust is restrained to rather high-level use cases by not exposing low-level CUDA API (such as thread or block index).

Similarly to Thrust, Kokkos [17] and RAJA [1] provide routines for common parallel idioms (`for_each`, `reduce`, `scan`) and they serve as portability layers for many systems such as HIP, OpenMP, CUDA or SYCL. However, they primarily focus on platform-agnosticism and do not provide the necessary abstractions for expressing traversal transformations.

7 Conclusion

We have presented a novel object-oriented approach for user-guided loop transformations focusing on the traversal of regular data structures. We base the abstraction on the Noarr library, expanding the Noarr paradigm for layout design to encompass loop transformations. This expansion significantly enhances the versatility of Noarr, enabling users to optimize memory access patterns by altering either the data structure layout, the traversal pattern, or both — all via a unified mechanism of applying composable first-class transformation objects. This approach promotes code independence (emphasizing separation of concerns) and reusability. It also simplifies semi-automated experimentation and performance tuning. Building the abstraction on top of Noarr (which automatically handles correct indexing and iteration ranges) further simplifies the transformation design process and makes it less prone to errors.

Besides the benefits related to memory access optimizations, the traverser abstraction is particularly useful for parallel processing. We demonstrate its utility with two implementation examples (TBB and CUDA) as proof of concept. Furthermore, we introduce an extension of Noarr that handles the management of replicated structures in CUDA shared memory. This functionality is particularly relevant in General-Purpose computing on Graphics Processing Units (GPGPU) programming.

Acknowledgments

This paper was supported by Charles University institutional funding SVV 260698 and Charles University Grant Agency (GAUK) project 269723.

References

[1] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. 2019. RAJA: Portable performance for large-scale scientific applications. In *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*. IEEE, 71–81.

[2] David Bednárek, Martin Kruliš, and Jakub Yaghob. 2021. Letting future programmers experience performance-related tasks. *J. Parallel and Distrib. Comput.* 155 (2021), 74–86.

[3] Nathan Bell and Jared Hoberock. 2012. Thrust: A productivity-oriented library for CUDA. In *GPU computing gems Jade edition*. Elsevier, 359–371.

[4] Lorenzo Chelini, Tobias Gysi, Tobias Grosser, Martin Kong, and Henk Corporaal. 2020. Automatic generation of multi-objective polyhedral compiler transformations. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 83–96.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[6] Philippe Clauss and Benoît Meister. 2000. Automatic memory layout transformations to optimize spatial locality in parameterized loop nests. *ACM SIGARCH computer architecture news* 28, 1 (2000), 11–19.

[7] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[8] Zhangxiaowen Gong, Zhi Chen, Justin Szaday, David Wong, Zehra Sura, Neftali Watkinson, Saeed Maleki, David Padua, Alexander Veidenbaum, et al. 2018. An empirical study of the effect of source-level loop transformations on compiler stability. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29.

[9] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 innovative parallel computing (InPar)*. Ieee, 1–10.

[10] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly: performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 04 (2012), 1250010.

[11] Junjie Li, Sanjay Ranka, and Sartaj Sahni. 2013. GPU matrix multiplication. *Multicore Computing: Algorithms, Architectures, and Applications* 345 (2013).

[12] Kedar S Namjoshi and Nimit Singhania. 2016. Loopy: Programmable and formally verified loop transformations. In *International Static Analysis Symposium*. Springer, 383–402.

[13] Chuck Pheatt. 2008. Intel® threading building blocks. *Journal of Computing Sciences in Colleges* 23, 4 (2008), 298–298.

[14] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[15] Adam Šmelko, Martin Kruliš, Miroslav Kratochvíl, Jiří Klepl, Jiří Mayer, and Petr Šimůnek. 2023. Astute Approach to Handling Memory Layouts of Regular Data Structures. In *Algorithms and Architectures for Parallel Processing: 22nd International Conference, ICA3PP 2022, Copenhagen, Denmark, October 10–12, 2022, Proceedings*. Springer, 507–528.

[16] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. 2010. Graphite two years after: First lessons learned from real-world polyhedral compilation. In *GCC Research Opportunities Workshop (GROW'10)*.

[17] Christian R Trott, Damien Lebrun-Grandie, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S Hollman, Dan Ibanez, et al. 2021. Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 805–817.

[18] Xingfu Wu, Michael Kruse, Prasanna Balaprakash, Hal Finkel, Paul Hovland, Valerie Taylor, and Mary Hall. 2022. Autotuning PolyBench Benchmarks with LLVM Clang/Polly loop optimization pragmas using Bayesian optimization. *Concurrency and Computation: Practice and Experience* 34, 20 (2022), e6683.