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.

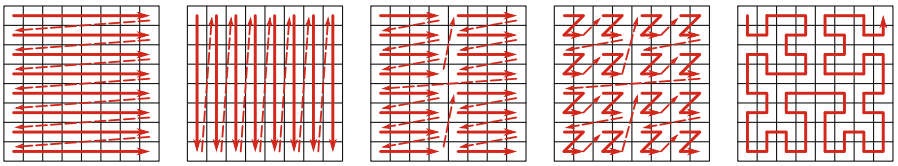
1.1 Motivational Example

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.

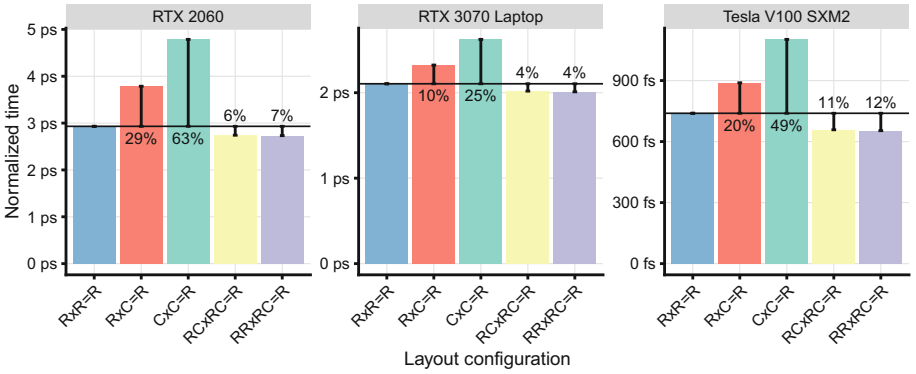


Fig. 2. Speedups of selected layout combinations relative to (row-major) baseline

```
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.

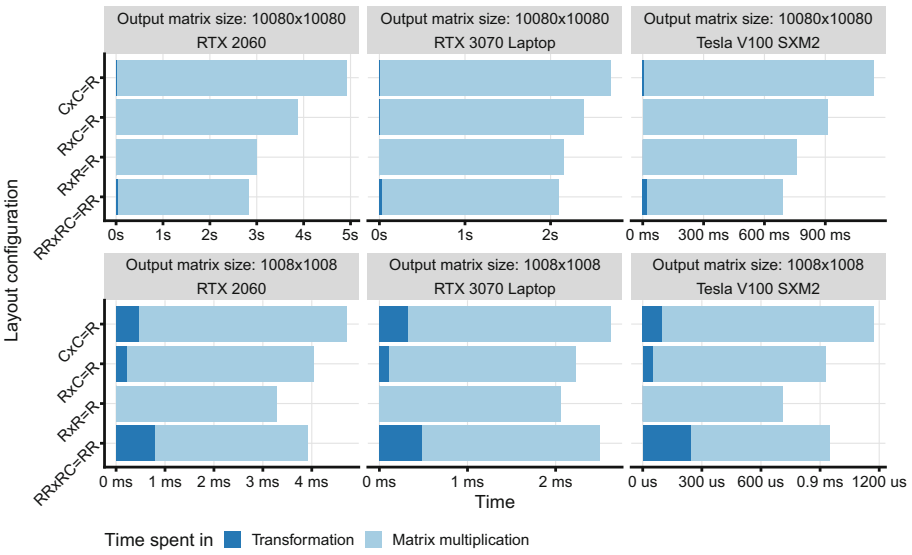


Fig. 3. Layout transformation times compared to actual matrix multiplication times

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.

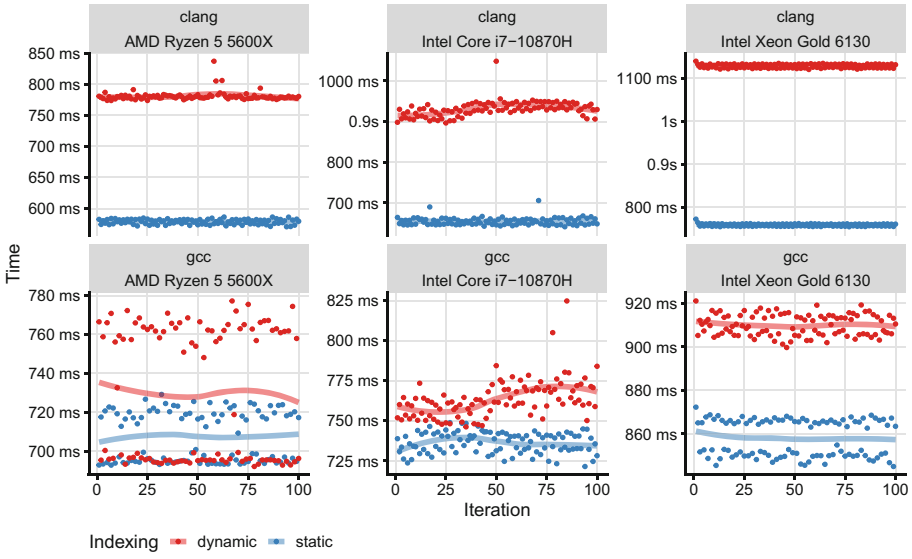


Fig. 4. Wall times of 100 stencil iterations (plotted lines represent the local regression of the measured times)

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.

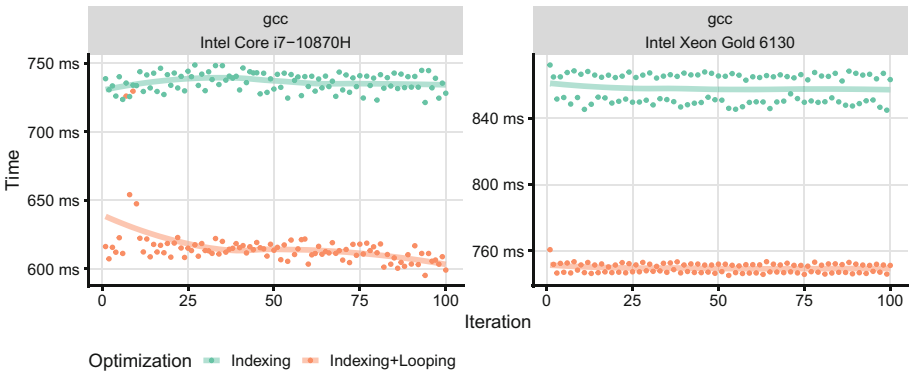


Fig. 5. Stencil execution times of two optimizations—compile-time *indexing* and the addition of constant-induced loop unrolling (*indexing+looping*)

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)