Parallel Computing 121 (2024) 103096

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Abstractions for C++ code optimizations in parallel high-performance applications

Jiří Klepl *, Adam Šmelko, Lukáš Rozsypal, Martin Kruliš

Department of Distributed and Dependable Systems, Charles University, Malostranské nám. 25, Prague, 118 00, Czech Republic

ARTICLE INFO

Keywords:
Regular data structure
Traversal
Plain C++
Parallel programming
Code optimization
Autotuning

ABSTRACT

Many computational problems consider memory throughput a performance bottleneck, especially in the domain of parallel computing. Software needs to be attuned to hardware features like cache architectures or concurrent 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 easily optimize for memory performance and employ semi-automated parallelization or autotuning without altering their internal code. We also add an abstraction for autotuning that allows defining tuning parameters in one place and removes boilerplate 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 or Intel DPC++. We evaluate the performance and expressiveness of our approach on the Polybench-C benchmarks.

1. Introduction

Memory operations affect the performance of applications in many ways. Contemporary CPUs dedicate a significant part of circuits (such as 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 latency of memory operations often depends on how the data are organized and how the memory is accessed. 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 (instructions being executed), 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.

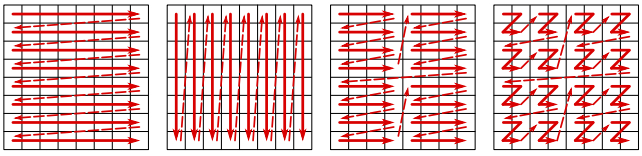
This paper focuses on regular data structures with multidimensional indexing (such as matrices, tensors, or grids). Such a data structure combines a multidimensional index space (i.e., dimensions), mapping into the linear space of offsets (layout), and address in the memory. The actual memory access pattern is then affected by the layout mapping and how the index space is traversed.

Let us illustrate the problem on a simple matrix with index space (i, j), where the indices run from 1 to H (height) and W (width), respectively. A matrix can be stored in many ways (Fig. 1) — e.g., in the traditional *row-major* layout, the linear offset is computed as $i \cdot W + j$. Traversing such a matrix by two nested loops (over i and j), the memory is accessed sequentially, which often performs optimally on contemporary CPUs. If we apply the same traversal to a matrix stored in a *column-major* layout with the offset computation $j \cdot W + i$, the subsequent memory operations are W elements apart, which disrupts the prefetching and may increase cache misses.

Transforming the layout of a data structure or the order of its traversal may have a profound effect on the performance [1]. Although the compilers attempt to optimize these operations (e.g., applying a polyhedral optimizer to reorder nested loops), these automated efforts do not always meet with optimal results since these optimizations are performed under imprecise assumptions about data dependencies and

* Corresponding author.
E-mail addresses: klepl@d3s.mff.cuni.cz (J. Klepl), smelko@d3s.mff.cuni.cz (A. Šmelko), lukas@rozsyp.al (L. Rozsypal), krulis@d3s.mff.cuni.cz (M. Kruliš).

https://doi.org/10.1016/j.parco.2024.103096
Received 6 May 2024; Received in revised form 10 July 2024; Accepted 12 August 2024
Available online 14 August 2024



(a) row-major (b) column-major (c) tiled row-major (d) z-curve

Fig. 1. Examples of common matrix layouts.

alignment. Furthermore, the code transformation search space is often vast, and the impact of a transformation on performance is difficult to predict. Manual transformations can lead to better performance, but designing such transformations may prove difficult, tiresome, and even error-prone, especially in parallel applications. Therefore, providing the programmer with code constructs for simple and flexible ways of expressing the desired transformations of traversal order might be beneficial.

We present an abstraction that facilitates a flexible specification of traversals over regular data structures. Our proposed implementation extends C++ library *Noarr*,[1] which provides first-class objects for defining memory layouts [2], in two ways:

1. We applied the layout abstraction of Noarr for loop transformations (*Noarr Traversers*).
2. We designed an autotuning adapter (*Noarr Tuning*) to simplify the tuning of layout or traversal parameters with existing tools (like OpenTuner).

The proposed solution has several benefits over contemporary libraries and tools that address the same problem. Most notably, we aim for standard C++ compilers so that our solution requires no compiler plugins or DSL preprocessing. The transformations are described using first-class objects, and their type is composable from pre-implemented (templated) classes provided by the library. This highly promotes code agnosticism (layouts and traversal orders are abstracted) as well as code reuse (the same transformations can be applied in different situations). Finally, we aim specifically at high-performance parallel applications providing direct connections to existing frameworks (like TBB, OpenMP, or CUDA).

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 discussed benefits are considered irrelevant. For instance, using a specific DSL may be easier in simple cases (Halide [3]) 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 traversal abstraction is explained in Section 3, and Section 4 describes its utilization for parallel programming (TBB, OpenMP and CUDA). Section 5 shows an extension of the abstraction for autotuning. The evaluation of performance and coding complexity is presented in Section 6. The related work is overviewed in Sections 7 and 8 concludes the paper.

2. Background

Specifying memory layouts and traversal orders (and their transformations) can be tackled using various approaches (besides the automated optimizations performed by the compiler):

- *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 [4]), 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 [3].

We investigate the native approach, aiming to step beyond traditional design patterns and software engineering practices. This involves exploiting the possibilities of the C++ language to its limits using templates, functional-like assembly of objects, and static (compile-time) meta-programming.

2.1. Noarr structures

We base our approach on the Noarr Structures library [2] that provides an expressive and flexible abstraction for creating data structure layouts. The key idea is constructing the layout via composing predefined template *proto-structures* such as arrays, vectors, or tuples. The following example shows two representations of a matrix — row-wise (`rw`) and col-wise (`cw`), depicted in Figs. 1(a) and 1(b), respectively. Let us emphasize the arguments `'i'` and `'j'` that identify the matrix dimensions — a characteristic feature of Noarr is that each dimension is uniquely identified by a name (usually a character).

```
auto rw = scalar<float>() ^ vector<'j'>() ^ vector<'i'>();
auto cw = scalar<float>() ^ vector<'i'>() ^ vector<'j'>();
```

The key point is that each structure is built from the bottom up. The process starts with a scalar and involves incremental applications of transformations via the `^` operator (e.g., applying `vector` adds a new dimension). Each such transformation produces a new (usually more complex) structure that can be used to create further structures. The following example demonstrates creating a matrix with a fixed size:

```
size_t size = 42;
auto matrix = rw ^ noarr::set_length<'i', 'j'>(size, size);
```

In this case, the matrix size is set at runtime and stored in the newly created object; however, using the same syntax, we can set the size at compile-time by replacing `size` with `noarr::lit<42>`, storing the size in the type.

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 memory offsets that can be used to access values in a data buffer:

```
size_t offset = matrix | noarr::offset<'i', 'j'>(i, j);
float &ref = matrix | noarr::get_at<'i', 'j'>(data, i, j);
```

This syntax for indexation applies to any Noarr structure that has the dimensions `'i'` and `'j'`, offering a unified way to access data in a layout-agnostic manner, which is crucial for our approach. The `|` operator applies a *getter* (on the right) to a structure (on the left). The following example shows retrieving the size of the matrix and the length of the `'i'` dimension:

```
size_t size = matrix | noarr::get_size();
size_t length = matrix | noarr::get_length<'i'>();
```

Concerning memory management, Noarr provides a wrapper called *bag* that binds the Noarr structure with a memory pointer. A binding with a raw pointer defines a flexible view into (borrowed) memory; a binding with a unique pointer defines ownership of the memory.

[1] https://github.com/ParaCoToUl/noarr-structures.

```
auto matrix_view = noarr::bag(matrix, data);
auto matrix_container = noarr::bag(matrix);
float &ref = matrix_view[noarr::idx<'i', 'j'>(i, j)];
```

The `noarr::idx<'i', 'j'>(i, j)` object (called *state*) represents a point in the index space of the matrix. It can be used to access an element of any `bag` with the covered indices.

2.2. Running examples

We describe two well-known algorithms by detailing their trivial implementations and elaborate on their potential for parallelization and optimization. They will be used as *running examples* to demonstrate the benefits of our approach.

2.2.1. Matrix multiplication

It presents one of the most profound and well-studied problems with many applications. We consider the naïve $\mathcal{O}(N^3)$ 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 matrix `C` can be computed independently (even concurrently), and the internal dot products are both associative and commutative. Typical optimizations are based on tiling, which requires splitting the outer two loops and may also enable efficient parallel processing [5].

2.2.2. Histogram

An approximation of the distribution of numeric data often used in data analysis, 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:

```
H[bins] = { 0, ... , 0 };
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 [6]. 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 (like in the case of 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).

3. Traversal abstraction

The (*Noarr Traversers*) abstraction is implemented as an extension of the C++ library Noarr by applying the same fundamental Noarr approach of specifying data layouts (and their transformations) via a composition of first-class objects (*proto-structures*) to the transformations of traversal orders.

A *traverser* is an object representing an index space and its corresponding traversal order constructed from one or multiple Noarr structures to be traversed together. The default index space unifies the dimensions of the provided structures. The user can then provide a callable object (like a lambda) specifying the action performed for each point of the index space. The individual points are represented by a *state* object that can index the appropriate elements of the traversed structures.

To alter the traversal order of the index space, a transformation structure can be applied to the traverser, producing a new traverser. The transformation structure is assembled from elemental first-class proto-structures like Noarr structures. The proto-structures represent loop transformations such as dimension interchange, tiling, z-curve, or even more general transformations such as introducing new loops, binding some iteration dimensions to specific indices, or restricting their spans.

When the dimensions of the traversed structures match entirely, the traverser represents a loop nest on these dimensions. In other cases, it can be used to perform various reductions, complex joins, or more general algorithms like matrix multiplication (explored further in this section) if the user provides input structures with appropriate dimensions.

The automatic traversal and indexation provided by the traverser abstraction enables the design of traversal-agnostic algorithms. With transformations defined as separate objects and used to produce altered traversers, we can create multiple versions of the same computation by applying different transformations to the same traverser without altering the internal code of the algorithm, offering an interface for dynamic code optimizations or semi-automated autotuning, which we discuss more in Section 5.

A traverser can also be used as an argument to a parallel executor, which then performs the traversal in parallel (we present ones based on TBB, OpenMP, and CUDA as examples in Section 4). The traversal is parallelized along one or multiple dimensions of the traverser, and each started thread is provided with an *inner traverser* representing the traversal of its corresponding section of the index space that is usually constructed via binding some dimensions to specific value ranges.

3.1. Introducing syntax for traversers

The syntax for traversers extends the syntax of Noarr Structures. Both these abstractions follow the pattern

```
base ^ transformationA ^ transformationB ^ ... | callable
```

where `base` is a scalar (in Noarr Structures) or a call to the `noarr::traverser` constructor (in Noarr Traversers). In both cases, this base object is transformed by the sequence of proto-structures and then used as an input for a callable object. The `callable` is a getter for a specific property of the structure or an element offset (in case of Structures) and a lambda function or a parallel executor (in case of Traversers). Furthermore:

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 proto-structure is applied to the base index space via the `^` operator. It changes the traversal order of the individual points of the index space. Suppose the original index space was equivalent to that of a structure `S`, after applying the transformation `T`, the index space is equivalent to that of the structure `S ^ T`.
3. The traverser is supplied via the `|` operator to a lambda function executed for each point of the index space represented by a state object. Alternatively, the traverser can be used as an argument to a parallel executor, offering a unified interface for parallel traversal over the index space, like ranges in the standard library.

When constructed using a single structure and supplied to a lambda function, the traverser iterates through the dimensions of that structure and calls the lambda function with a *state* object that represents a

point in the index space. It 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) | [=](auto state) { c[state] = 0; };
```

The traverser automatically creates a unified index space, combining the spaces of input structures by preserving a unique instance of each dimension. Depending on which dimensions the structures have in common, the resulting index space may range from identical to the space of each structure (complete unification) up to a cartesian product of all dimensions (no unification). By naming the indices of three matrices $\mathtt{a}(i,k)$, $\mathtt{b}(k,j)$, and $\mathtt{c}(i,j)$, the index-unification process allows us to write even more complex algorithms like matrix multiplication simply as:

```
noarr::traverser(a, b, c) | [=](auto state) {
    c[state] += a[state] * b[state];
};
```

The traverser extracts dimensions i,k from the first structure and then j from the second (all other dimensions are duplicates), which yields the index space to be the cartesian product of (i,k,j). In other words, the index space corresponds to the three perfectly nested loops of the naïve matrix multiplication algorithm presented in Section 2.2.

The traversal order of its index space can be transformed by applying a transformation proto-structure. 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 below.

```
auto tiles = noarr::into_blocks<'i', 'I'>(noarr::lit<16>) ^
             noarr::into_blocks<'k', 'K'>(noarr::lit<16>) ^
             noarr::into_blocks<'j', 'J'>(noarr::lit<16>) ^
             noarr::hoist<'I', 'J', 'K'>();

noarr::traverser(a, b, c) ^ tiles | [=](auto state) {
    c[state] += a[state] * b[state];
};
```

The example shows a transformation structure composed of multiple proto-structures. The `into_blocks` proto-structure splits the given dimension into two dimensions, one representing the local index within the block and the other representing the block index. The `hoist` proto-structure then moves the block indices to the outermost traversal loops. The parameter `noarr::lit<16>` ensures that the block size is represented as a compile-time constant.

Many transformations are already implemented in the Noarr library, including renaming/reordering the indices, restricting iteration spans, fixing indices in given dimensions to specific values, and some more complex operations designed for parallel processing. Details are provided in our replication package [2]

3.2. Traversal over sections of the index space

In some situations, iterating sections of the index space instead of single values may be required (representing an imperfect loop nest). A typical example is accumulating a portion of the dot product corresponding to a given block in a register to reduce the number of memory operations. In such cases, we put the lambda within the `for_dims` wrapper, which takes a list of dimensions representing the traversed sections. The lambda is then executed for each section of the index space and provided with *inner traverser* for traversal over the internal section (using the same traverser interface).

[2] https://github.com/jiriklepl/ParCo2024-artifact.

```
noarr::traverser(a, b, c) ^ tiles
    | noarr::for_dims<'I', 'J', 'K', 'j', 'i'>(
        [=](auto inner_trav) {
        auto res = c[inner_trav]; // local var (register)
        inner_trav | [=, &res](auto state) {
            res += a[state] * b[state];
        };
        c[inner_trav] = res;
    });
```

4. Parallel execution

We use the histogram running example (Section 2.2) as a representative of a simple parallel reduction problem. First, let us show its sequential implementation using Noarr traversers:

```
auto in = bag(scalar<char>() ^ vector<'i'>(size), i);
auto out = bag(scalar<size_t>() ^ array<'v', 256>(), o);
noarr::traverser(in) | [=](auto state) {
    out[noarr::idx<'v'>(in[state])] += 1;
};
```

The algorithm iterates over the input data in the `in` bag consisting of `size` characters and increments the corresponding bin in the histogram stored in the `out` bag. The `noarr::idx<'v'>` function maps the input values in their histogram bins.

In Noarr, parallel execution is implemented via specialized executors that take a traverser as an argument and implement the calls to the given platform. This approach separates the concerns of the traverser and the parallel execution, making the code more modular and extensible. We implemented executors for TBB [7], OpenMP [8], and CUDA [9] as representatives of different technologies to demonstrate modularity. A demonstration of an executor for parallel reduction using TBB follows:

```
noarr::tbb_reduce(
    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` executor takes five arguments: the traverser, the output bag, and three lambda expressions. The first lambda initializes the output structure, the second performs the element-wise reduction, and the third merges the privatized histogram copies. The reduction is performed automatically over the whole space defined by the traverser and parallelized along its first dimension. The user can specify a different dimension for parallelization by applying the `hoist<Dim>()` transformation to the traverser.

The parallel executor transparently privatizes the given output structure to prevent data collisions when the parallelized dimension does not parametrize the output structure (like in the example) and, thus, different threads can access the same memory. Then, the executor creates a local copy of the output structure for each worker thread as needed (managed by `tbb::combinable`). The third lambda defines a merging procedure for two copies of the histogram.

For the simple case of parallel *for-each* (which can be used for the histogram computation if the privatization is handled manually), we also present parallel executors for standard C++ parallelization and OpenMP. All of these share the same interface, so the user can easily choose between them, as presented in the following example of a parallel histogram computation with explicit privatization. Initialization and merging of the privatized copies are omitted for brevity.

```
// manually split the input into chunks
auto traverser = noarr::traverser(in ^
    noarr::into_blocks<'i', 't'>(BLOCK_SIZE));
// privatize the output structure for each chunk
auto priv_out = noarr::bag(out_struct ^
    noarr::vector_like<'t'>(traverser.get_struct()));
auto task = [&](auto state) {
    priv_out[state + noarr::idx<'v'>(in[state])] += 1;
};

// use one of the following parallel executors:
noarr::tbb_for_each(traverser ^ noarr::hoist<'t'>(), task);
noarr::omp_for_each(traverser ^ noarr::hoist<'t'>(), task);
noarr::std_for_each(traverser ^ noarr::hoist<'t'>(), task);
```

In this example, we split the input data into chunks of size `BLOCK_SIZE` and allocate a private copy of the output structure `out` for each chunk. The `task` lambda performs the same operation as in the previous examples. `vector_like` creates a vector of the same size as the corresponding dimension of the traversed structure, and `hoist` is used to move the `'t'` dimension to the outermost loop so that the computation is performed in parallel over the chunks. The `tbb_for_each`, `omp_for_each`, and `std_for_each` functions execute the `task` lambda in parallel using TBB, OpenMP, and standard C++, respectively.

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 [9] or SYCL [10], which employ custom compilers that extend but remain compatible with C++ language. We present an adaptor for CUDA as a proof of concept, but a similar approach can be used for SYCL.

CUDA framework is based on the SIMT (Single Instruction, Multiple Threads) paradigm. CUDA threads are spawned collectively (forming a *grid*) executing a single piece of code (*kernel*). Each thread is given index structures (`threadIdx`, `blockIdx`) to identify a data element processed by the thread. Threads are grouped into *thread blocks* so they can cooperate more closely (via on-chip *shared memory* or using faster synchronization primitives). The indexing structures (for threads and blocks) can encompass up to three dimensions to conveniently deal with multidimensional data (like matrices or 3D grids).

The proposed CUDA traverser wraps a regular traverser describing how its dimensions are mapped to the grid. This mapping is subsequently used to generate the grid execution parameters (block size and block count) and internal traverser that can be used in the kernel. The following code represents a kernel that computes the histogram (stored in global memory) using atomic updates (a typical implementation), where each thread computes multiple input values. The aggregation of work per thread is a necessary optimization (which we discuss further in the paper) that significantly decreases global memory writes (and atomic collisions).

```
template<class InTr, class In, class Out>
__global__ void histogram(InTr in_trav, In in, Out out) {
    in_trav | [=](auto state) {
        auto value = in[state];
        atomicAdd(&out[noarr::idx<'v'>(value)], 1);
    };
}
```

The `in_trav` is an inner traverser created from the traverser of the input data in the kernel invocation to traverse a section of the data within a 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::bag(in_blk_struct, in_ptr);
auto out = noarr::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 some of the traverser dimensions with the CUDA grid dimensions. In this case, the `b` becomes the block index and `t` the thread index within the block. The resulting *cuda traverser* then provides kernel invocation parameters via its methods `grid_dim()` and `block_dim()`. The inner traverser `in_trav` passed as the first argument to the kernel is the result of binding `b` and `t` dimensions to the `blockIdx` and `threadIdx` CUDA 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 input traverser. This permits a certain level of agnosticism in the parallelization of algorithms. The composable nature of traversers makes it possible to separate the blocking operations required for CUDA execution into a predefined proto-structure that may be applied to a traverser as a transformation. Noarr also provides a `simple_run()` method as a shortcut since kernel execution is the most frequent operation.

4.2. Shared memory privatization

Massively parallel systems are particularly susceptible to intensive data synchronization. In the `histogram` kernel, the atomic updates may 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 *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 in the shared memory for each warp lane (32 copies per thread block). This way, threads in a warp have no collisions among themselves and the aggregation in the shared memory significantly decreases the number of global memory transactions. Afterward, the individual copies need to be merged into the final copy in the global memory before a thread block terminates.

The shared memory is divided into 32 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 In, class Shm, class Out>
__global__ void histogram(InT in_trav, In in, Shm shm_s, Out
    out) {
    extern __shared__ char shm_ptr[];
    auto shm_bag = noarr::bag(shm_s, shm_ptr);
    // initialize shared memory (zero the bins)
    in_trav | [=](auto state) {
        auto val = in[state];
        atomicAdd(&shm_bag[noarr::idx<'v'>(val)], 1);
    };
    // reduce shm copies of histogram into global memory
}
```

The `atomicAdd` uses the bag allocated in the shared memory, and the `shm_s` structure transparently handles access to private copies (based on the thread index). The `shm_s` structure is constructed externally in our example, so the kernel is more generic, and the shared memory utilization can be subjected to external tuning; however, it can also 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 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) ^ subset | [=](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) ^ noarr::cuda_step_block() | [=](auto
     state){
    size_t sum = 0;
    for (size_t i = 0; i < shm_s.num_stripes(); ++i) {
        sum += shm_bag[state + noarr::cuda_stripe_idx(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. Autotuning

The autotuning process is often governed by a script that searches for optimal parameters for the tuned program. Hence, the tuned parameters need to be declared twice — in the tuning script (with possible value domains) and in the tuned applications (where they affect the behavior of the program). The difficulty is keeping both sides up to date and ensuring that the parameters are properly passed into the compilation process (static parameters) or via command line arguments (dynamic parameters).

The core part of an example from our domain of selecting optimal traversal order for a data structure is listed below:

```
// define policy classes
struct traverse_by_rows { ... };
struct traverse_by_columns { ... };
struct traverse_by_blocks { ... };

auto policy = POLICY; // tuning parameter (passed as a macro)
for_each(policy, matrix, [](auto& value) { modify(value); });
```

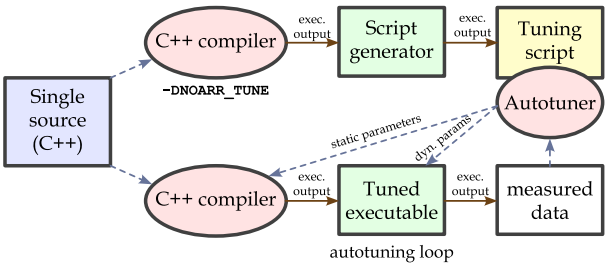


Fig. 2. Schema of the autotuning process with Noarr Tuner.

If this application is optimized by OpenTuner [11], for instance, the Python script governing the tuning has to properly inject argument `-DPOLICY` to the C++ compiler, and the value of this macro must match the identifier of one of the prepared `traverse_` structures. Any extension or modification of the traverse possibilities implemented in the C++ application must be duly reflected in the Python script, making the development process tedious and error-prone.

We propose an abstraction that allows the tuning parameters (and their domains) to be declared directly with the tuned application in a single-source manner. The tuning script is then generated from this source using our extension to the Noarr framework in a special compilation step (depicted in Fig. 2).

The compilation is controlled by a macro (`NOARR_TUNE`). If present, the code is compiled in a *generator* mode, which bypasses the `main` function of the application, creating an executable that only generates the script for the autotuner (e.g., a Python script for OpenTuner or Optuna [12]). In the case of a native C++ autotuner (ATF [13]), the compiled executable directly runs the tuning loop. Without the `NOARR_TUNE` macro, the compilation results in a *regular* executable that runs the application (which can be subjected to tuning).

5.1. Noarr tuning abstraction

To demonstrate the syntax and technical details of our Noarr Tuning abstraction, we will build on the matrix multiplication running example (Section 2.2). Even the naïve implementation of the algorithm provides many options for tuning, such as the memory layout of the matrices, blocking factor, or loop order.

Listing 1 presents specifying these choices in Noarr syntax. It defines a static `tuned` object that represents the tuned parameters. It is used both to generate the tuning script (if compiled in generator mode with `NOARR_TUNE` macro) and to preprocess and handle parameters passed from the autotuning framework (if compiled regularly).

The following list describes the macros used in the Noarr Tuning abstraction in Listing 1 and their behavior:

- `NOARR_TUNE_BEGIN(formatter)` specifies an autotuning backend together with a build process (e.g., direct execution of a specific compiler or CMake) and a measurement process (e.g., running the binary and collecting the reported wall clock time). The `formatter` is a placeholder for a specific backend configuration as described in Section 5.2.
  This macro generates the appropriate directives to initialize the autotuning backend in the generator mode. It takes no action in the regular compilation mode.
- `NOARR_TUNE_PAR` declares a tuned parameter and its domain. The first argument is the name of the parameter used to identify it on both sides (tuned code and the tuner). The remaining arguments define the type, allowed values, and other type-specific options.

```
using namespace noarr;
struct tuned {
    NOARR_TUNE_BEGIN(formatter);

    NOARR_TUNE_PAR(cLayout, tuning::choice,
        scalar<float>() ^ vector<'i'>() ^ vector<'j'>(),
        scalar<float>() ^ vector<'j'>() ^ vector<'i'>());
    // aLayout (i, k) and bLayout(k, j) defined analogically

    // pow(n) = 2 ** n: 1, 2, 4, 8, 16, 32
    NOARR_TUNE_PAR(blockSize, tuning::mapped_range, pow, 5);

    // xor_fold(a, b, c) = (a ^ b) ^ c
    NOARR_TUNE_PAR(loopOrder, tuning::mapped_permutation,
        xor_fold, hoist<'j'>(), hoist<'k'>(), hoist<'i'>());
    NOARR_TUNE_PAR(blockOrder, tuning::mapped_permutation,
        xor_fold, hoist<'J'>(), hoist<'K'>(), hoist<'I'>());

    NOARR_TUNE_CONST(blocks, into_blocks<'i', 'I'>(*blockSize) ^
                             into_blocks<'j', 'J'>(*blockSize) ^
                             into_blocks<'k', 'K'>(*blockSize));

    NOARR_TUNE_CONST(order, *blocks ^ *loopOrder ^ *blockOrder)
      ;

    NOARR_TUNE_END();
} tuned;

// algorithm:
auto C = bag(*tuned.cLayout ^ set_lengths<'i','j'>(I, J));
auto A = bag(*tuned.aLayout ^ set_lengths<'i','k'>(I, K));
auto B = bag(*tuned.bLayout ^ set_lengths<'k','j'>(K, J));
traverser(A, B, C) ^ *tuned.order | [=](auto state) {
    C[state] += A[state] * B[state];
};
```

Listing 1: Noarr Tuning abstraction for the naïve matrix multiplication

In the generator mode, the parameter generates the appropriate directives for the autotuning framework and assumes the default value for the given parameter type (e.g., for the `choice` type, the first value). In the regular compilation mode, the parameter assumes the value passed by the autotuning framework.

- `NOARR_TUNE_CONST` specifies a constant parameter usually derived from the other parameters. The first argument specifies the parameter's name and the second its value. The behavior of this macro is the same regardless of the compilation mode.
- `NOARR_TUNE_END` operates only in generator mode and finalizes the specification. It triggers the generator of the autotuning code (which usually writes the generated script to std. output) and then *stops the runtime* by calling `std::exit(0)`, thus suppressing the `main` function.

The parameters specified in the `tuned` object can be accessed in the algorithm implementation using the dereference operator `*` (e.g., `*tuned.blockSize`). This applies to parameters and constants in both compilation modes.

Since the `tuned` object is defined in the static scope, it is initialized before the program starts; hence, the `END` macro can stop the runtime before the main algorithm executes (in the generator mode), ensuring that the algorithm runs only in the regular compilation mode without applying any changes to the algorithm implementation.

The described mechanism reduces the boilerplate code for the autotuning process since it is automatically generated by the Noarr Tuning abstraction. The single-source approach simplifies the autotuning process, reduces the possibility of errors, and promotes the separation of concerns and the reusability of the code (as demonstrated in the matrix multiplication example, where the tuning specification is separated from the algorithm implementation).

5.2. Autotuning backend

Noarr Tuner was designed modularly to work with any autotuner. The `formatter` placeholder in Listing 1 selects the autotuning backend and specifics of the build process and algorithm execution on the given platform. As an example, Listing 2 demonstrates a formatter specification for OpenTuner.

```
using namespace noarr::tuning;
opentuner_formatter(
  std::cout, // output stream for the autotuning script
  cmake_compile_command_builder(PRJ_PATH, BUILD_DIR, "algorithm"
     ),
  direct_run_command_builder(BUILD_DIR / "algorithm"),
  MEASUREMENT_COMMAND);
```

Listing 2: OpenTuner formatter specification for the Noarr Tuning abstraction

Each subsequent tuning macro in Listing 1 calls a `format` method of the formatter that registers the parameter using its name and value specification. The value specification is an object representing the parameter type and the possible values. Another level of abstraction is added to the value representation since the backend does not need to know the specific values of the parameter (e.g., choice lists are converted to numbers).

The OpenTuner formatter specified in Listing 2 uses the *CMake compiler command builder* and *Direct run command builder* wrappers to generate the commands for recompiling the algorithm and running the binary, respectively. This allows the abstraction to support further build systems and measurement processes. The `MEASUREMENT_COMMAND` is a string that specifies the command used by the OpenTuner backend to measure the performance of the algorithm, which allows the user to specify the fitness metric and how it is retrieved from the algorithm runtime.

Consider the tuning problem from Listing 1 with OpenTuner formatter and a `CMakeLists.txt` file that specifies the project setup with targets `algorithm` and `algorithm-tune` representing the two compilation modes. The targets are distinguished by defining `NOARR_TUNE` for the `algorithm-tune` target (triggering the generator mode). The following commands then run the whole autotuning process:

```
mkdir -p build && cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
cmake --build . -t algorithm-tuning

# run the OpenTuner script
python3 <(./algorithm-tuning) --test-limit=NUM_TESTS
```

The specification of the autotuning backend via a formatter allows us to easily extend the Noarr Tuning abstraction to support other autotuning frameworks. The only requirement is to implement the formatter for the given backend supporting calls to the `format(name, value_specification)` method with the appropriate value specifications.

5.3. Noarr tuning in the context of parallelism

In Section 4, we discuss the parallelization of algorithms by passing a Noarr Traverser representing an indexation space with a given iteration order to a parallel executor that wraps the desired parallelization framework. The executor handles one or more dimensions of the iteration in parallel by spawning tasks (like TBB) or a grid of threads (like CUDA). To demonstrate the synergy of both proposed Noarr abstractions, we build on the examples from Section 4 and combine them with autotuning, allowing us to present more coherent examples of Noarr Tuner.

The histogram running example (Section 2.2) demonstrated parallelization that employed the CUDA framework. This problem is particularly interesting since it has several parameters that affect performance (providing good candidates for tuning) — the block size, the number of elements processed by each thread, and the number of private copies in the shared memory. The first two are used when re-shaping the index

```
struct tuned {
  NOARR_TUNE_BEGIN(formatter);

  NOARR_TUNE_PAR(blockSize, noarr::tuning::choice, 256, 512, 10
     24);
  NOARR_TUNE_PAR(elemsInThread, noarr::tuning::choice, 1, 2, 4,
      8);
  NOARR_TUNE_PAR(privCopies, noarr::tuning::choice, 1, 2, 4);

  NOARR_TUNE_CONST(grid,
    noarr::into_blocks<'i','B','t'>(*blockSize) ^
    noarr::into_blocks<'B','b','x'>(*elemsInThread));

  NOARR_TUNE_END();
} tuned;

template<class InT, class In, class Shm, class Out>
__global__ void histogram(InT in_trav, In in, Shm shm_s, Out
     out) {
  __shared__ int shm_ptr[];
  auto shm_bag = noarr::bag(shm_s, shm_ptr);
  // initialize shared memory (zero the bins)
  in_trav | [=](auto state) {
    atomicAdd(&shm_bag[noarr::idx<'v'>(in[state])], 1);
  };
  // reduce shm copies of histogram into global memory
}

void run_histogram(auto in, auto out) {
  auto ct = noarr::cuda_threads<'B', 't'>(
    noarr::traverser(in ^ *tuned.grid));
  auto shm_s = out.structure() ^
    noarr::cuda_striped<*tuned.privCopies>();

  histogram<<<ct.grid_dim(), ct.block_dim(),
    shm_s | noarr::get_size()>>>(ct.inner(), in, shm_s, out);
}
```

Listing 3: Noarr Tuning use for the CUDA framework

space, and the last one is supplied to the `noarr::cuda_striped<N>` construct to ensure bank-aware distribution of the private copies in the shared memory. Listing 3 presents the example of CUDA histogram from Section 4.2, adapted for autotuning.

```
struct tuned {
  NOARR_TUNE_BEGIN(formatter);
  NOARR_TUNE_PAR(loop_order, noarr::tuning::choice,
    noarr::hoist<'i', 'j'>(), noarr::hoist<'j', 'i'>());
  NOARR_TUNE_END();
} tuned;

void matrix_multiplication(auto A, auto B, auto C) {
  noarr::tbb_for_each(
    noarr::traverser(C),
    [=](auto state) { C[state] = 0; });

  noarr::tbb_for_each(
    noarr::traverser(A, B, C) ^ *tuned.loop_order,
    [=](auto state) { C[state] += A[state] * B[state]; });
}
```

Listing 4: Noarr Tuning use for the TBB framework

The example in Listing 4 builds on the TBB implementation of matrix multiplication ($C := A \times B$) from Section 2.2. In this case, the tuning process should select the best dimension for parallel processing. The `tuned` object specifies the loop order of the matrix multiplication algorithm via hoisting the given dimension to the topmost position. The selected executor (`tbb_for_each`) is used to run the algorithm in parallel along the hoisted (topmost) dimension.

The two presented examples show a selection of several tuning choices for different parallelization frameworks. Together with Listing 1, they demonstrate the flexibility and expressiveness of the Noarr Tuning abstraction that is extensible to different parallelization frameworks by specifying the appropriate executor wrapper and also to different autotuning backends by implementing the appropriate formatter. The abstraction is designed to work well with the Noarr Structures and Noarr Traversers abstractions, but it can be used independently as well to tune any (hyper-)parameters of the algorithm or the parallelization framework.

6. Evaluation

The evaluation has the following objectives: (1) We would like to demonstrate that the proposed abstraction has no additional performance overhead. (2) We discuss its qualities from the perspective of the programmers using multiple code complexity metrics to assess its added complexity and perform a comparison with other approaches (annotations, DSL). (3) We evaluate the autotuning capabilities of the abstraction and its compilation overhead. We have also considered performing a user study, but that is currently beyond the scope of this paper and we were unable to find a sufficient number of volunteers.

Due to space limitations, we present only selected key results in the paper. A complete set of experiments and results is available in the replication package [3]

6.1. Methodology and datasets

The presented performance evaluations were measured on Intel Xeon Gold 6130 (CPU) and Tesla V100 PCIe 16G (GPU) compiled with GCC 13.2.0 and NVCC 12.3. Each test comprised one warmup run and 10× subsequent measured runs. The wall time of the tested kernel was measured by a high-resolution system clock. We present only mean values whenever the variance of the measured times is very low (below 1%); otherwise, we present the boxplots (with 25%, 50%, and 75% quantiles).

We used *Polybench/C-4.2.1*[4] and *Polybench/GPU-1.0*[5] [14] benchmark suites (using the `EXTRALARGE` dataset) 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 (e.g., 2DConvolution). For Polybench/GPU, we have implemented 5 algorithms as a representative subset for the evaluation.

6.1.1. Code complexity metrics

To assess the complexity of the Noarr implementation compared to the baseline C code, we use a set of metrics that capture the textual and semantic complexity of the code. The presented metrics are measured collectively on all kernels of the Polybench suite unless stated otherwise. However, we also provide results that aggregate their measurements on individual kernels in the artifact. We discuss cases that show a significant difference between these two granularities.

For the basic textual comparison, we use the number of lines of code, the total number of characters, and the number of individual code tokens (as identified by `clang` lexer) as metrics. Additionally, we measure the size of `gzip`-compressed code to get an idea of the entropy of the code (for this metric, the size of each kernel compressed separately and the size of the archive containing all kernels are both presented, since the complexity estimates are expectedly different for these two cases).

To capture the semantic complexity of the code, we use the *McCabe cyclomatic complexity* [15] and *Halstead complexity* [16] measures based on the control flow and the number of unique operators and operands in the code, respectively. The McCabe complexity counts the control

[3] https://github.com/jiriklepl/ParCo2024-artifact.
[4] https://sourceforge.net/projects/polybench/files.
[5] https://github.com/sgrauerg/polybenchGpu.

flow statements (occurrences of `if` and `for` and lambda expressions replacing the `for`-loops are the only control flow constructs present in compared code) and adds the number of connected control flow components (+1 for each kernel). The Halstead complexity measures use counts of the operators and operands tokenized by the `clang` lexer — operands consist of non-function identifiers (mostly local variables) and literals (primarily numbers), operators consist of other tokens.

The aforementioned metrics are not specifically intended for capturing the complexity of indexing data structure or how difficult it is to transform its traversal order. For an illustration, transposing a data structure and tiling its traversal with two-dimensional indexation `A[i][j]` will require expressions like `A[J*TSIZE + j][I*TSIZE + i]` with a similar change to the traversal's loop bounds and step sizes. Where Noarr code would use much simpler `A[state]` expression (as the `state` contains the appropriate indices regardless of the transformation), and the corresponding traverser does not explicitly specify any bounds.

To capture the transformability, we require metrics focusing on code constructs that direct the traversal and indexation of the data structures and are potentially affected by their transformations. In our domain, these constructs are the indexation expressions and loop statements (or their equivalents, such as traversers). Thus, we define two additional metrics:

- The *indexation complexity* counts loop statements and all occurrences where a data structure is indexed (an element is being accessed).
- The *subscript complexity* is a more fine-grained metric that counts individual loop expressions (loop bounds and step sizes) and the number of indexing sub-expressions (i.e., every bracket used for indexing).

For instance, expression `A[i][j] = B[j][i]` has indexation complexity 2 and subscript complexity 4. This approach is inspired by other presented metrics which measure a linear count of some textual or semantic elements in the code. The most direct comparison is the McCabe complexity — branches into specific parts in a procedure are analogous to the indexation expressions accessing parts of data, and the procedures are analogous to the loops in the code.

A traversal with lower indexation complexity is likely easier to transform, as fewer constructs must be changed. However, the proposed metrics do not measure the complexity of the expressions used for indexation (like `J*TSIZE + j`) since similar expressions are usually simpler or the same in Noarr implementations compared to the C baseline. Thus, our metrics provide a conservative estimate of the described complexity related to the indexing of data structures.

6.1.2. Threats to validity

The greatest concern is whether our Noarr implementation is directly comparable with the original Polybench code since we rewrote the algorithms into the Noarr manually. 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 a `noarr::vector`.
2. The loops from the baseline implementation are directly mapped to Noarr traverser abstractions.
3. Kernels are structurally equivalent, and their computation statements are in the same order and rewritten into an equivalent form.
4. Data structure accesses happen at the equivalent computation points and access elements in the same order.
5. The time measurements and device synchronizations (for GPU) take place at equivalent program points.

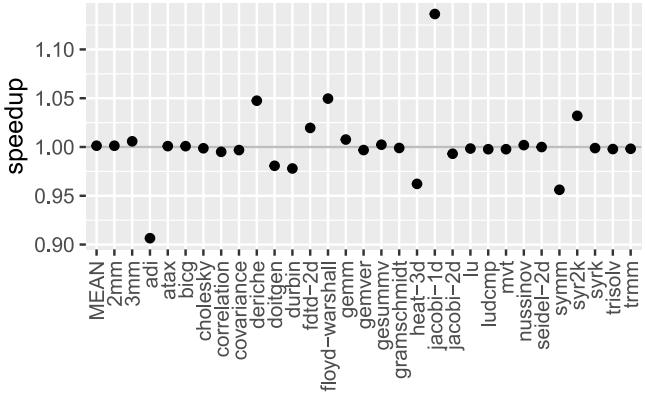


Fig. 3. Comparing Noarr to plain C on Polybench/C-4.2.1.

Rewriting the algorithms according to these requirements is not easily automatable, nor can we automatically verify that all the rules have been upheld. As a basic precaution, we at least included scripts that check whether the implementations produce the same result. With additional code cross-reviews, we are reasonably certain that the corresponding C and Noarr codes represent the same algorithm with identical implementation optimization, only written in different abstractions.

The code complexity comparisons are also threatened by the fact that the source codes were written manually. To mitigate this problem, we extract each kernel from the Polybench suite and the corresponding Noarr implementation delimited by the `scop` pragma that is used to mark the kernel code (usually for polyhedral optimizations). The boilerplate code (running the kernels, measuring their execution times, and dumping the results), which is shared among all algorithms in each implementation, is removed from the measurements since it is not subject to any transformations and does not depend on the algorithm. The code that initializes the input data structures is not subject to any transformations either (transforming it could taint its use for verification); therefore, it is removed as the boilerplate.

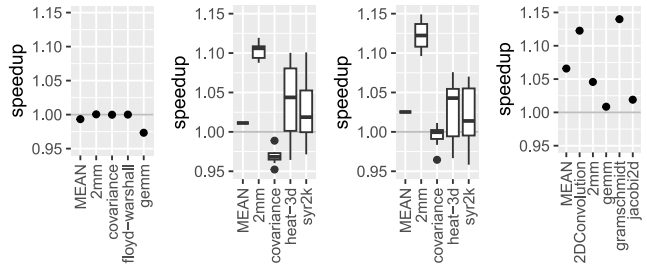
After the extraction, we use `clang-format` to ensure that the code is formatted consistently (among all kernels and between the two implementations) for the text-based metrics to be as accurate as possible. The only affected metrics are the lines of code, character count, and the compressed code size, as the other metrics are independent of the formatting.

6.2. 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 obfuscated the code for the compiler and prevented some optimizations.

Fig. 3 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 five outliers where Noarr performed better and five 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. The `MEAN` column shows the geometric mean of all speedups (1.00×).

Fig. 4(a) presents the speedups of a selected subset of Polybench algorithms that were subjected to hand-tuning (applying tiling and loop reordering). Figs. 4(b) and 4(c) present the results of selected algorithms with their outermost loop in the critical segment parallelized by TBB and OpenMP, respectively. Finally, the GPU results (using CUDA



(a) Hand-tuned serial (b) Parallel: TBB (c) Parallel: OpenMP (d) Parallel: CUDA

Fig. 4. Comparing selected algorithms Noarr vs. baseline.

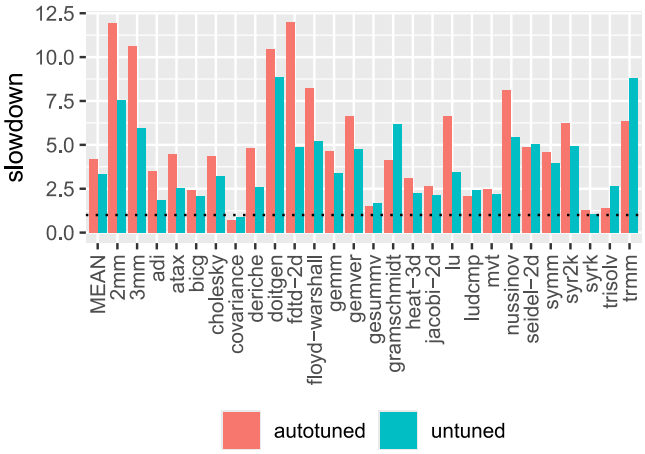


Fig. 5. Compilation times of the reimplemented algorithms compared to the baseline C implementation.

traverser) are presented in Fig. 4(d). The results indicate that the additional traverser transformations applied in Noarr and the parallelization extensions do not have significant overhead over direct implementation in C, TBB, OpenMP, 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 each graph for TBB and OpenMP parallelization (4(b) and 4(c)). Each plot also includes a `MEAN` column with the geometric mean of all speedups for the algorithms in the given category.

6.3. Autotuning results

To evaluate the capabilities of the Noarr Tuning abstraction, we have added our proposed autotuning mechanism to the Noarr implementation of algorithms from the Polybench/C suite — some of which have been slightly modified to allow for more extensive tuning. For the experiment, we used OpenTuner backend (we tested all implemented backends, but it has no bearing on these results).

In Table 1, we present the ten algorithms that allow for arbitrary loop reordering and tiling on some of the dimensions. The table shows the number of loop reorders, tile sizes, tile layouts, and data layouts — all of which are different choices for the autotuner to select from. Since each choice affects the actual data structure types and loop orders in the algorithm, and only a small portion of that is usually explored until an optimum is reached, it would not be sensible to compile all possible combinations in a single binary; for some of the algorithms, it would not be even feasible on many platforms due to the vastness of the search space (the size of the search space is presented on the next row). It is thus important to recompile the algorithms in the autotuning process for the actual parameters chosen by the autotuning backend.

The last four rows of the table show the number of parameters that the autotuner can adjust (total number, number of ranges, number of categories, and number of permutations). It shows that some of the presented algorithms include non-trivial combinations of the discussed parameter types.

The speedups of the autotuned algorithms and histories of the autotuning process are subject to the used autotuning backend and are not presented in this paper as the focus is on automatization of presenting the tuning search space to the autotuning backend and recompiling the code with the selected parameters — not on the actual tuning process itself. However, the Noarr Tuning abstraction hides the specifics of the tuning backend, so it can be easily replaced with any other, possibly better-performing, backend.

Fig. 5 shows the mean compilation (wall clock) times of the direct reimplementation (`untuned`) and the recompiled code during autotuning (`autotuned`) visualized as a slowdown in comparison to the baseline C implementation for the given algorithm; accompanied by the geometric `MEAN` of the slowdowns (`untuned`: 3.34×, `autotuned`: 4.16×). Recompilation uses precompiled headers to speed up the process (as only the algorithm code changes), and the tuned parameters are passed via a file instead of arguments to the compiler. The presented results show that the Noarr Traverser abstraction has a significant overhead in compilation time. However, this overhead increases only slightly with the tuning abstraction despite the increase in complexity of the algorithm.

The compilation time overhead increase (relative to the runtime) of the recompiled code compared to the baseline C compilation (which represents the minimum estimate equivalent to replacing all tuning abstraction with preprocessor macros) is −6.7% for `covariance` (min), 12 660% for `atax` (max), and 997.7% on average (median: 23.22%). For the algorithms presented in Table 1, the compilation time overhead increase is 1.64% for `floyd-warshall` (min), 3077% for `gemver` (max), and 534.7% on average (median: 31.3%). The measurements are extremely volatile due to the non-deterministic selection of paths chosen by the autotuning backend.

6.4. Comparison to alternate approaches

Comparing loop transformation approaches from the code design perspective is difficult 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 7. We use the matrix multiplication running example optimized for memory transfers by blocking.

For each tool, we present an equivalent code example that implements the matrix multiplication computation kernel and specifies the intended loop transformation (tiling and loop reordering). We focus solely on the code differences and not on any performance differences since such would be due to independent factors such as further optimizations or implementation details of the tools.

```
1  float A[I][K], B[K][J], C[I][J];
2
3  for (i = 0; i < I; i++)
4      for (j = 0; j < J; j++)
5          for (k = 0; k < K; k++)
6  mul:        C[i][j] += A[i][k] * B[k][j];
7
8  // in a separate file:
9  affine(mul, {[i,j,k]->[i,k,j]})
10 affine(mul, {[i,k,j]->[oi,oj,ok,ii,ij,ik]: oi=[i/32] and ii=
       i%32 and oj=[j/32] and ij=j%32 and ok=[k/32] and ik=k%3
       2})
```

Listing 5: Loopy (using `affine` directives)

Table 1
Autotuning results for selected Noarr implementations of Polybench/C-4.2.1 algorithms.

Algorithm	2mm	3mm	symm	trmm	floyd-warshall	gemver	mvt	doitgen	heat-3d	jacobi-2d
Loop reorders	4	8	2	2	6	8	4	2	6	2
Tile sizes	7^4	7^6	7^2	7^2	7^3	7^6	7^4	7^2	7^3	7^2
Tile layouts	4	8	2	2	6	8	4	2	6	2
Data layouts	32	128	8	4	2	2	2	12	4	4
Search space size	1.23 M	0.96 G	1.57 k	0.78 k	32.9 k	7.53 M	76.8 k	2.35 k	49.4 k	0.78 k
Number of parameters	13	19	7	6	6	13	9	6	7	6
- Integral ranges	4	6	2	2	3	6	4	2	3	2
- Categories	9	13	5	4	1	7	5	3	0	4
- Permutations	0	0	0	0	2	0	0	1	4	0

Listing 5 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 mul{"mul"};
Halide::Var i{"i"}, j{"j"};
Halide::RDom k{0, K};

mul(i, j) = C(i, j); // Initial values
mul(i, j) += A(i, k) * B(k, j); // Matrix multiplication

Halide::Var ii{"i_inner"}, ij{"j_inner"};
Halide::RVar ik{"k_inner"};

mul.update().tile(i, j, ii, ij, 32, 32).split(k, k, ik, 32)
            .reorder({i, j, k, ii, ij, ik});
mul.realize(C);
```

Listing 6: Halide (DSL using methods on function stages)

The Halide implementation (Listing 6) 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 code 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'>(32) ^
                into_blocks<'j', 'J'>(32) ^
                into_blocks<'k', 'K'>(32) ^
                hoist<'I', 'K', 'J', 'i', 'j', 'k'>();

traverser(A, B, C) ^ my_order | [&](auto state) {
    C[state] += A[state] * B[state];
};
```

Listing 7: Native C++ with Noarr traversers

Finally, Listing 7 presents an 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 to 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, simplifying the design of similar data structures and the optimization of similar algorithms. Furthermore, this code can be compiled by any C++ compiler without extra preprocessing.

6.5. Code complexity

In this section, we discuss the complexity of the Noarr implementation compared to the baseline C code according to the metrics described in Section 6.1.1. The goal is to assess the implementation overhead of the Noarr abstraction and the potential benefits it brings to the programmer.

All the measured metrics are presented in Table 2. Most textual metrics (characters count, code tokens, and gzipped code) suggest about a 20% increase in the textual complexity (i.e., verbosity) of the code due to the Noarr abstraction. This result is expected as the Noarr constructs have generally longer names than the C constructs they replace — this is best visible on the contrast between the code tokens and characters count metrics, where the Noarr code has only 6.4% more individual tokens, but 21.7% more characters. The lines of code metric contrast this trend, showing a decrease of 13.1% in the Noarr implementation, which is primarily due to the baseline C code containing a higher number of explicit iterations (loop statements in C, traversers in Noarr) — each appearing on a separate line.

The McCabe complexity, which assesses the control flow, shows an average decrease of 19% in the Noarr implementation, which is an expected result as Noarr Traversers can substitute multiple nested loops — this is consistent with the lines of code. The Halstead difficulty metric gives a result consistent with the textual metrics, showing an increase of 20% in the Noarr implementation, while the Halstead effort metric shows an increase of 35%.

The indexation complexity and subscript complexity metrics show 36.8% and 70.5% decrease in the Noarr implementation, respectively. This shows how much redundant data layout and traversal information Noarr can abstract away on the two granularities discussed in Section 6.1.1. The decrease of the explicit indices and bounds also gives an estimation of the increase in the layout and traversal agnosticism of the code that enables the programmer to design the algorithm in such a way that it can be easily optimized by transforming the data structures and traversals without changing the algorithm itself. In Noarr, this constitutes applying the transformation objects to the traversers and data structures that comprise the composition of the desired transformations.

7. 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 [17] or Polly in LLVM [18]. 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 [19] that addressed the loop transformations from the perspective of optimizing memory operations is over 20 years old. Since then, several models based on static predictions have been created [17,18]. The latest innovations focus on elaborate multi-objective scheduling for loop transforms [20].

Table 2
Comparison of code complexity metrics between the baseline C code and the Noarr code (negative difference means that Noarr is better).

Metric	Baseline C	Noarr	Difference
Lines of code	452	393	−13.1%
Characters count	14 424	17 558	+21.7%
Code tokens	6157	6551	+6.4%
Gzipped code size	6011	7447	+23.9%
Gzipped archive size	3446	4115	+19.4%
McCabe complexity	189	153	−19.0%
Halstead length	6127	6521	+6.4%
Halstead vocabulary	156	205	+31.4%
Halstead volume	44 637.7	50 077.9	+12.2%
Halstead difficulty	308.6	371.6	+20.2%
Halstead effort	$1.38 \cdot 10^7$	$1.86 \cdot 10^7$	+34.9%
Indexation complexity	525	332	−36.8%
Subscript complexity	1126	332	−70.5%

Autotuning methods address the optimization problem by generating variations of the 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. [21] presented a tuning tool based on Polly [18] that employs Bayesian optimizations.

7.1. Domain specific languages

Many works address the issue of separating the specifics of memory access patterns and traversals from the algorithm by defining the algorithm via some DSL with a simplified model that facilitates applying various transformations. We have selected Halide language [3] as a state-of-the-art representative with a superficially similar approach. Although Halide is designed for optimized image processing, their approach found use in deep learning algorithms [22] as well. However, their approach relies on a custom compilation pipeline and a runtime library, while Noarr depends solely on the C++ language.

The Halide language [3] uses a decoupling approach similar to our transformation of traversers via proto-structures. In Halide, a *schedule* defines traversal transformations and parallelization on the defined algorithms. However, the schedules lack the extensibility and composability of proto-structures.

7.2. Annotations

Another approach employed by various tools and compiler extensions uses code annotations that specify the desired layout and loop transformations. The Loopy [4] system, perhaps closest to our research, extends the LLVM compiler and provides custom affine transformations and testing for the legality of loop transformation.

7.3. Native tools

The projects closest to our approach can be characterized by being built using native C++ abstractions and thus allowing for more seamless interaction with other C++ features, intrinsics, or user-defined abstractions and avoiding the necessity for custom development toolkits in favor of existing tools for C++ development, significantly 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 multidimensional data layouts. The `mdspan` class template can express many layouts of multidimensional arrays but does not provide a way of expressing traversals and their transformations.

The NVIDIA Thrust library [23] provides routines for parallel code execution on both CPU and GPU. It is a header-only C++ library. While offering 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 by not exposing low-level CUDA API (such as thread or block index).

Similarly to Thrust, Kokkos [24] and RAJA [25] provide routines for common parallel programming 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.

The CuTe library by the CUTLASS [26] project is a header-only library that resembles our approach. It provides a layout abstraction for regular data structures expressed as a multilinear mapping and a *tensor* wrapper that combines a given layout with some data. These abstractions are similar to Noarr Structures (more specifically, proto-structures and their `bag` wrappers) in expressiveness and general use. This similarity is most noticeable in defining more complex layouts via layout algebra that performs functional composition of the different layout objects. Their approach allows for this by defining a linearized index space for each layout, while Noarr proto-structures are specifically designed to be equivalent to partial functions on index spaces. However, the CuTe library is focused almost exclusively on tensor representations and does not provide a way to express traversal transformations.

8. Conclusion

We presented a novel abstraction for user-guided loop transformations focusing on the traversal of regular data structures. We base the abstraction on the Noarr library and its 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 via a unified mechanism of applying composable first-class transformation objects. In addition, we introduced an abstraction for autotuning that allows the programmer to define tuned parameters directly in the tuned application and automatically generates necessary control scripts for the autotuner.

Besides the benefits related to memory access optimizations, the traverser abstraction is particularly useful for parallel processing. We demonstrate its utility with several examples (TBB, OpenMP, 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.

Presented abstractions promote code independence (separation of concerns) and reusability. They also simplify semi-automated experimentation and performance tuning. Building the abstraction on top of Noarr (which automatically handles correct indexing and ranges) further simplifies the transformation design process and makes it less error-prone. In addition, the abstraction can be used to statically gather behavioral information from the code which can help to guide the autotuning process with next-generation methods like deep learning.

Extension summary

This article extends the paper *Pure C++ Approach to Optimized Parallel Traversal of Regular Data Structures* [27] presented in the proceedings of The 15th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM 2024). This extension introduces the novel abstraction for autotuning (Section 5)

and refines the syntax for traversers (Section 3). The demonstration of the utility of traversers in parallel programming (Section 4) was extended to discuss OpenMP parallelization and a unified syntax for the parallelized for-each loop construct across multiple parallel backends to demonstrate its extensibility. The evaluation (Section 6) was extended to cover the new autotuning abstraction, compilation times, and purely syntactic complexity metrics that indicate how difficult the abstraction may be for programmers. Most of the remaining sections were rewritten to accommodate the aforementioned changes as well as to incorporate feedback we received.

CRediT authorship contribution statement

Jiří Klepl: Writing – original draft, Visualization, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Adam Šmelko:** Writing – review & editing, Writing – original draft, Conceptualization. **Lukáš Rozsypal:** Writing – review & editing, Software, Conceptualization. **Martin Kruliš:** Writing – review & editing, Supervision, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This paper was supported by the Johannes Amos Comenius Program (P JAC) project No. CZ.02.01.01/00/22_008/0004605 (Natural and anthropogenic georisks), by Charles University institutional funding SVV (grant number 260698), and Charles University Grant Agency (GAUK, grant number 269723).

References

[1] Z. Gong, Z. Chen, J. Szaday, D. Wong, Z. Sura, N. Watkinson, S. Maleki, D. Padua, A. Veidenbaum, et al., An empirical study of the effect of source-level loop transformations on compiler stability, Proc. ACM Program. Lang. 2 (OOPSLA) (2018) 1–29.

[2] A. Šmelko, M. Kruliš, M. Kratochvíl, J. Klepl, J. Mayer, P. Šimůnek, 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, 2023, pp. 507–528.

[3] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, S. Amarasinghe, Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines, Acm Sigplan Notices 48 (6) (2013) 519–530.

[4] K.S. Namjoshi, N. Singhania, Loopy: Programmable and formally verified loop transformations, in: International Static Analysis Symposium, Springer, 2016, pp. 383–402.

[5] J. Li, S. Ranka, S. Sahni, GPU matrix multiplication, in: Multicore Computing: Algorithms, Architectures, and Applications, Vol. 345, CRC Press, 2013.

[6] D. Bednárek, M. Kruliš, J. Yaghob, Letting future programmers experience performance-related tasks, J. Parallel Distrib. Comput. 155 (2021) 74–86.

[7] C. Pheatt, Intel® threading building blocks, J. Comput. Sci. Coll. 23 (4) (2008) 298.

[8] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1) (1998) 46–55.

[9] D. Guide, Cuda C Programming Guide, vol. 29, NVIDIA, 2013, p. 31, July.

[10] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, X. Tian, Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL, Springer Nature, 2021.

[11] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, S. Amarasinghe, Opentuner: An extensible framework for program autotuning, in: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, 2014, pp. 303–316.

[12] T. Akiba, S. Sano, T. Yanase, T. Ohta, M. Koyama, Optuna: A next-generation hyperparameter optimization framework, in: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2019, pp. 2623–2631.

[13] A. Rasch, M. Haidl, S. Gorlatch, ATF: A generic auto-tuning framework, in: 2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems, HPCC/SmartCity/DSS, 2017, pp. 64–71, http://dx.doi.org/10.1109/HPCC-SmartCity-DSS.2017.9.

[14] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, J. Cavazos, Auto-tuning a high-level language targeted to GPU codes, in: 2012 Innovative Parallel Computing, InPar, Ieee, 2012, pp. 1–10.

[15] T.J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. SE-2 (4) (1976) 308–320.

[16] M.H. Halstead, Natural laws controlling algorithm structure? ACM Sigplan Not. 7 (2) (1972) 19–26.

[17] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, R. Upadrasta, Graphite two years after: First lessons learned from real-world polyhedral compilation, in: GCC Research Opportunities Workshop, GROW'10, 2010.

[18] T. Grosser, A. Groesslinger, C. Lengauer, Polly: performing polyhedral optimizations on a low-level intermediate representation, Parallel Process. Lett. 22 (04) (2012) 1250010.

[19] P. Clauss, B. Meister, Automatic memory layout transformations to optimize spatial locality in parameterized loop nests, ACM SIGARCH Comput. Archit. News 28 (1) (2000) 11–19.

[20] L. Chelini, T. Gysi, T. Grosser, M. Kong, H. Corporaal, Automatic generation of multi-objective polyhedral compiler transformations, in: Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques, 2020, pp. 83–96.

[21] X. Wu, M. Kruse, P. Balaprakash, H. Finkel, P. Hovland, V. Taylor, M. Hall, Autotuning PolyBench Benchmarks with LLVM Clang/Polly loop optimization pragmas using Bayesian optimization, Concurr. Comput.: Pract. Exper. 34 (20) (2022) e6683.

[22] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al., {TVM}: An automated {end-to-end} optimizing compiler for deep learning, in: 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 18, 2018, pp. 578–594.

[23] N. Bell, J. Hoberock, Thrust: A productivity-oriented library for CUDA, in: GPU Computing Gems Jade Edition, Elsevier, 2012, pp. 359–371.

[24] C.R. Trott, D. Lebrun-Grandie, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D.S. Hollman, D. Ibanez, et al., Kokkos 3: Programming model extensions for the exascale era, IEEE Trans. Parallel Distrib. Syst. 33 (4) (2021) 805–817.

[25] D.A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A.J. Kunen, O. Pearce, P. Robinson, B.S. Ryujin, T.R. Scogland, RAJA: Portable performance for large-scale scientific applications, in: 2019 Ieee/Acm International Workshop on Performance, Portability and Productivity in Hpc, P3hpc, IEEE, 2019, pp. 71–81.

[26] V. Thakkar, P. Ramani, C. Cecka, A. Shivam, H. Lu, E. Yan, J. Kosaian, M. Hoemmen, H. Wu, A. Kerr, M. Nicely, D. Merrill, D. Blasig, F. Qiao, P. Majcher, P. Springer, M. Hohnerbach, J. Wang, M. Gupta, CUTLASS, 2023, URL https://github.com/NVIDIA/cutlass.

[27] J. Klepl, A. Šmelko, L. Rozsypal, M. Kruliš, 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.