Pure C++ Approach to Optimized Parallel Traversal of Regular Data Structures

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

Abstract

Many computational problems consider memory throughput a performance bottleneck. The problem becomes even more pronounced in the case of parallel platforms, where the ratio between computing elements and memory bandwidth shifts towards computing. Software needs to be attuned to hardware features like cache architectures or memory banks to reach a decent level of performance efficiency. This can be achieved by selecting the right memory layouts for data structures or changing the order of data structure traversal. In this work, we present an abstraction for traversing a set of regular data structures (e.g., multidimensional arrays) that allows the design of traversal-agnostic algorithms. Such algorithms can be adjusted for particular memory layouts of the data structures, semi-automated parallelization, or autotuning without altering their internal code. The proposed solution was implemented as an extension of the Noarr library that simplifies a layout-agnostic design of regular data structures. It is implemented entirely using C++ template meta-programming without any nonstandard dependencies, so it is fully compatible with existing compilers, including CUDA NVCC. We evaluate the performance and expressiveness of our approach on the Polybench-C benchmarks.

CCS Concepts: • **Computing methodologies → Parallel programming languages**; **Parallel algorithms**; • **Software and its engineering** → *Object oriented development*; *Compilers.*

PMAM '24, March 03, 2024, Edinburgh, UK
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Keywords: iteration, traverser, memory optimizations, parallel, order agnostic

ACM Reference Format:
Jiří Klepl, Adam Šmelko, Lukáš Rozsypal, and Martin Kruliš. 2018. 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 (PMAM '24).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Memory operations are a cause of bottleneck in many situations. Contemporary CPUs dedicate a significant part of their circuits (such as multi-level caches or prefetching units) to mitigate this problem. In parallel processing, the situation becomes even more complicated as some resources are shared by the cores (like L3 cache, memory controllers, or memory buses), and the memory transactions need to be kept coherent (by MESI protocol, for instance). GPUs introduce another level of complexity caused by the lockstep execution model where multiple threads perform the exact instruction in the same cycle (so the memory transactions need to be planned across multiple cores) and by introducing special memory types like shared memory (with concurrently accessible banks).

The performance of many programs is often heavily affected by how they access data in the memory. If the data dependencies permit, the operations accessing the memory can be (re)arranged to take advantage of caching, prefetching, coalesced loads, parallel memory banks, or concurrent utilization of memory controllers without affecting the semantics (i.e., the results) of the algorithm. Even when the (re)arrangement does not change the number of operations, it may reduce the execution time if the latencies of the data transfers decrease. Unfortunately, the optimal arrangements are often system-specific and rather difficult to find.

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

Let us illustrate the problem on a common matrix. It defines the index space (i, j), where the dimensions run from 1 to H (height) and W (width) respectively. A matrix can be stored in many ways (Figure 1). Perhaps the most common is the *row-major* order, where linear offsets are computed as $i \cdot W + j$. If the matrix is traversed by two nested for-loops (over i and j), the memory will be accessed sequentially, which often performs optimally on contemporary CPUs. If we swap the loops (j will become the outer loop), the subsequent memory operations will be W elements apart, which disrupts the prefetching and may increase cache misses.

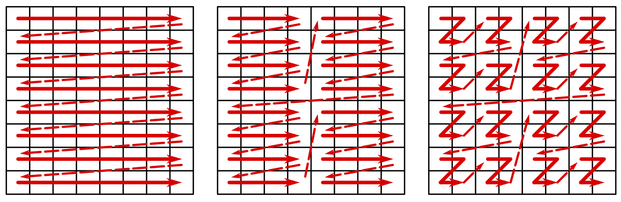
(a) row-major **(b)** tiled **(c)** z-curve

Figure 1. Examples of common matrix layouts

Transforming the layout of a data structure or the order of its traversal may have a profound effect on the performance [8]. Although the compilers attempt to optimize these operations (e.g., by application of polyhedral optimizer to reorder nested loops), these automated efforts do not always meet with success since the compilation is bound with strict assumptions about data dependencies and alignment, the transformation search space is vast, and it is often difficult to predict the impact a transformation has on performance. Designing such transformations manually may prove difficult, tiresome, and even error-prone, especially in the domain of parallel applications. Therefore, it might be beneficial to provide the programmer with code constructs that would allow for explicit yet simple and flexible ways of expressing the desired transformations of traversal order.

In this work, we present an abstraction that facilitates a flexible specification of traversals of regular data structures. Our proposed implementation is an extension of C++ library *Noarr*[1], which provides first-class structures for defining memory layouts [14]. Our extension (*Noarr Traversers*) uses the same design philosophy (templated first-class transformation structures) for semi-automated traversal (over the indexing space) and provides basic transformation elements such as loop interchange, strip-mining, tiling, or z-curve. The proposed solution has the following benefits over contemporary libraries and tools that aim at the same problem:

1. *Standard compilers support*: The abstraction is defined in standard C++ and does not require any compiler extensions or domain-specific language (DSL) preprocessing, which is usually the case with annotation-based and DSL-based frameworks such as Loopy [11] or Halide [13].
2. *First class transformations*: A transformation is assembled from prepared templated classes and instantiated as a first-class object that is then applied in an algorithm written using traversal-agnostic loop constructs. This promotes code reusability (multiple versions of an algorithm are produced by applying different transformations), and it also allows constant parameters to be embedded in the type, moving some of the computation into compile time.
3. *Custom transformations*: The user has the expressive power of an imperative language (C++) to define custom transformations, not being limited by the syntax of annotation-based frameworks or restricted DSLs.
4. *Suitable for parallelism*: The proposed framework is designed to be easily utilized on various parallel platforms and libraries, namely multicore CPUs (TBB) and manycore GPUs (CUDA).

The aforementioned benefits should simplify coding when dealing with manual optimizations. More importantly, we aim to create an ecosystem where this abstraction can be used for semi-automated optimizations using autotuning or machine learning models. In such systems, designing the code in a traversal-agnostic way (or the data structures in a layout-agnostic way) simplifies the injection of the layouts or traversal patterns by the external optimizer. Furthermore, the flexibility of the proposed abstraction allows for choosing the desired trade-offs between the performance and the complexity of the compilation due to meta-programming (e.g., by embedding the constant parameters in the type).

Let us emphasize that the aforementioned benefits define the intended group of users for our tool. Other approaches may be better (lead to faster implementations or require less code to write) in cases where some of the benefits are considered irrelevant. For instance, using a specific DSL may be easier in simpler cases (Halide [13]) at the cost of universality and the necessity for more compilation steps.

The paper is organized as follows. Section 2 explains Noarr and introduces the running examples. The proposed abstraction is explained in Section 3, and Section 4 describes its utilization for parallel programming (TBB and CUDA). Section 5 presents the evaluation results. The related work is summarized in Section 6 and Section 7 concludes the paper.

2 Background

The problem of memory layouts and traversal order of data structures can be tackled using various approaches (besides the automated optimizations performed by the compiler):

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

- *Native approach* uses only native constructs of the selected language. In C++, for instance, class policies can be used for selecting data structure layouts and iterators for data structure traversal.
- *Annotations* may be introduced into the language to hint to the compiler how the data structures (e.g., arrays) or loops may be transformed. This approach usually builds on native compiler optimizations (e.g., to guide polyhedral optimizer [11]), 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 [13].

We investigate the native approach; however, our objective is to step beyond the traditional design patterns and software engineering practices. We aim at exploiting the possibilities of C++ language to its limits using templates, functional-like assembly of data types, and static (compile-time) metaprogramming.

2.1 Noarr structures

We base our solution on the Noarr library [14], which provides an abstraction for creating data structure layouts. The key idea is that the layout is represented by a first-class structure. The type of a Noarr structure is assembled from predefined templated base types like arrays, vectors, or tuples. The following example shows two representations of a matrix — row-wise (`rw`) and col-wise (`cw`). Let us emphasize the arguments `'i'` and `'j'` which identify the dimensions.

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

The two structures define the abstract layout of a matrix. The structures are immutable, and each can be used as a basis for creating various structures with fixed sizes, for example:

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

In this case, the matrix size is set at runtime, and so the size is stored in the object; however, using the same syntax can embed the sizes in the type so they are computed at compile-time (for example, if `size` was `noarr::lit<42>`).

Another important principle of Noarr is decoupling the layouts from memory management. The structures used in the previous examples have no binding to memory. They represent an indexing abstraction for computing linear offsets, which can be used in internal and external data structures alike, or it can be used with any base pointer to dereference memory values:

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

Since most data structures reside in the main memory, Noarr offers a wrapper called *bag*, which binds the Noarr structure with a pointer. If we do not specify a memory location for the data represented by the Noarr structure, the bag automatically allocates the memory on the heap. However, the user can also specify a memory location (e.g., a memory-mapped file or a shared memory in the CUDA kernel), and the bag will use that instead.

```
auto matrix = noarr::make_bag(matrix_struct);
float &ref = matrix.template at<'i', 'j'> (i, j);
```

The current implementation of the bag does not employ any more complex memory management operations like host-device memory transfers or memory mapping. Such operations need to be controlled by the user of Noarr. The bag merely specifies indexing semantics on top of a pointer. However, extending this abstraction to a more complex behavior in the future is technically possible.

2.2 Running examples

In this section, we detail two running examples that we will use to demonstrate the syntax and the benefits of the proposed abstraction in the later sections.

2.2.1 Matrix multiplication. It presents one of the most profound problems with many applications. Being a well-studied problem, we can draw on the known optimizations and express them using our abstractions. We rely on the naïve $O(N^3)$ algorithm, which computes elements of the output matrix as dot products. Having square matrices `A` and `B` (of the size N^2), the product matrix `C` may be computed as:

```
for (size_t i = 0; i < N; ++i) {
    for (size_t j = 0; j < N; ++j) {
        C[i][j] = 0;
        for (size_t k = 0; k < N; ++k) {
            C[i][j] += A[i][k] * B[k][j];
}   }   }
```

The individual elements of the output matrix can be computed independently (even concurrently), and the internal dot products are both associative and commutative, allowing more fine-grained optimizations. Typical optimizations are based on tiling, which requires splitting the outer two loops and may also enable efficient parallel processing [10].

2.2.2 Histogram. An approximation of the distribution of numeric data often used in data analysis and related fields (e.g., machine learning or similarity search). The objective is to assign data elements into predefined bins (categories) and count the number of elements in each bin. Having histogram `H` and a function that finds a bin for each element, the algorithm can be coded simply as:

```
for (size_t i = 0; i < N; ++i)
    H[findBin(data[i])] += 1;
```

The histogram algorithm is particularly interesting from the perspective of parallel computing [2]. When the input elements are processed concurrently, the histogram updates must be synchronized (e.g., by atomic instructions). If the number of bins is low and the level of concurrency high (typically on a GPU), the histogram updates will become a bottleneck. In such cases, sophisticated methods of privatization (and subsequent merging of private copies) could be beneficial. Another perspective is that a histogram can be computed as a bin-wise parallel reduction (with per-bin data filtering). These issues will help us to demonstrate the capabilities of the proposed abstraction.

3 Proposed Abstraction

We propose an abstraction for flexible traversal of regular data structures. This abstraction is implemented as an extension (named *Noarr Traversers*) of the C++ library Noarr. The extension applies the fundamental Noarr approach of specifying data layouts via a composition of elemental first-class objects (called *proto-structures* in Noarr) to the transformations of traversal orders.

A *traverser* is a first-class object that represents an index space and its corresponding traversal order. It is constructed from one or multiple Noarr structures to be traversed together. The traverser constructs the base index space from the combination (unification) of dimensions of the provided structures. The user can then provide a callable object (usually a lambda expression) that specifies the action performed on elements indexed by each point of the index space. For a single structure, this corresponds to the for-each algorithm. For two structures presenting the same set of dimensions (but not necessarily the same layout), the traverser can be used, for example, to copy the values from one structure to the other, which can implement transposition — this generalizes to other common algorithms such as reduction if we transform the dimensions of the input structures accordingly.

To alter the traversal order of the index space, the traverser can be transformed by applying a transformation structure, producing a new traverser. The transformation structure is assembled from elemental first-class proto-structures in a similar way to Noarr structures. The proto-structures defined for this purpose represent basic loop transformations such as loop interchange, strip-mining, tiling, z-curve, or more general transformations such as introducing new loops, binding some iteration dimensions to specific indices, or restricting their spans. The transformation structure can be defined separately and reused for different traversers.

Transforming the traversers by applying a separate object from the outside enables a simple way to design traversal-agnostic algorithms. We can then create multiple versions of the same computation by applying different transformations to the same traverser.

A traverser can also be used as an argument to a parallel executor, which then performs the traversal in parallel (we have implemented one based on TBB and one for CUDA, as examples). The parallelization is guided by one or multiple dimensions of the traverser, and each started thread is provided with an *inner traverser* representing the traversal of its corresponding traversal section that is usually constructed via binding some dimensions to specific values.

3.1 Introducing syntax for traversers

A traverser is constructed and executed in three steps that also denote the three key principles:

1. The constructor of the traverser is given one or more Noarr structures and deduces the *base index space* from them by unifying their dimensions.
2. A transformation is applied to the base index space via the `.order(transformation)` method. It changes the traversal order of the individual points of the index space. This step is optional and possibly reoccurring (composing the provided transformations into one).
3. The `.for_each(action)` method is called, where the actual body of the traverser (usually a lambda function) is injected. This provides a uniform interface that can be used for sequential iteration and parallel processing.

When constructed using a single structure, the traverser iterates through the cartesian product of the dimensions of that structure and calls the provided lambda function (body) with a tuple-like *state* object. The state object represents a point in the index space that can be used to access the corresponding element of the traversed structure. The following example performs an element-wise initialization of structure `c` (like a traditional for-each algorithm):

```
noarr::traverser(c).for_each([=](auto state) { c[state] = 0; });
```

The traverser can properly combine the indexing space from multiple Noarr structures by creating a cartesian product of different dimensions while unifying matching dimensions based on their names (template identifiers). If we name indices of three matrices $\mathtt{a}(i, k)$, $\mathtt{b}(k, j)$, and $\mathtt{c}(i, j)$ the matrix multiplication can be written simply as:

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

The traverser extracts dimensions i, k, k, j, i, j, which (after unification) yields the indexing space to be the cartesian product of (i, k, j). In other words, the index space corresponds to the three nested loops of the naïve matrix multiplication.

The traverser and its index space can be transformed using the `order()` method, which takes a transformation structure as an argument. In the case of matrix multiplication, the most common transformation would be to perform tiling — i.e., splitting each of the indices into an index of a block (of fixed size) and a local index within the block. An example of such transformation is presented in the following.

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

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

Note that the transformation structures can be declared separately so they can be reused for different traversers (and vice-versa). The `strip_mine` template performs *tiling* where the first index denotes the dimension to be tiled, and the second two denote the newly created dimensions (existing dimensions are replaced). The tiling is followed by *hoisting*, which moves the first of the two created dimensions into the outermost traversal loop. The `noarr::lit<16>` ensures the constant tile size is embedded into the type.

In some situations, it is beneficial to iterate over whole sections of the index space instead of single values. A typical example of that is accumulating a portion of the dot product corresponding to a given block in a local variable (a register) to reduce the number of memory operations. In such cases, we replace `for_each` call with templated `for_dims`, which takes a list of dimensions that represent the sections to be traversed. It creates an instance of an *inner traverser* corresponding to the current index space section. The inner traverser offers the same traverser interface, so it can be used for an internal traversal over the given section without changing the body of the traversal.

```
noarr::traverser(a, b, c)
  .order(blocks)
  .template for_dims<'I', 'J', 'K', 'j', 'i'>(
    [=](auto inner_trav) {
      auto res = c[inner_trav.state()];
      inner_trav.for_each([=, &res](auto state) {
        res += a[state] * b[state];
      });
      c[inner_trav.state()] = res;
  });
```

There are many transformations already implemented in Noarr. That includes renaming and reordering the indices, restricting iteration spans and slicing, fixing indices in particular dimensions, and some more complex operations designed for parallel processing. Details can be found in our replication package[2].

4 Parallel Execution

Besides the benefits granted by the iteration order agnosticism of traversers, the abstraction can easily be extended to parallel processing. A parallel *for-each* example would be trivial, so we start with parallel reduction.

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

The demonstration is based on the histogram running example (Section 2.2). The sequential implementation (presented above) comprises a simple for-loop. The `in` variable is a bag (a wrapper that combines Noarr structure with memory pointer `i`) holding the input (vector of `char`) and `out` is a bag holding the histogram (256 bins stored in `o`).

We have decided to design the parallel executors as external tools that take a traverser as an argument instead of extending the traverser interface. This approach is more modular and can be easily extended by implementing new parallel executors using various libraries (C++ standard library, TBB [12], or OpenMP [7]). As a proof of concept, we present a TBB implementation of the parallel reduce algorithm wrapper for Noarr traversers.

```
noarr::tbb_reduce_bag(
    noarr::traverser(in),
    [](auto out_state, auto &out_left) {
        out_left[out_state] = 0;
    },
    [in](auto in_state, auto &out_left) {
        out_left[noarr::idx<'v'>(in[state])] += 1;
    },
    [](auto out_state, auto &out_left, const auto &out_right) {
        out_left[out_state] += out_right[out_state];
    },
    out);
```

The `tbb_reduce_bag` algorithm template takes five arguments. Besides the traverser and the output bag, there are three lambdas — the first initializes the output structure to zero, the second performs the element-wise reduction, and the third performs the merging of privatized copies of the output structure (histogram).

The reduction is performed automatically over the whole space defined by the traverser, but only the first dimension is processed concurrently. The user can explicitly change which dimension is the first by applying `.order` to the traverser, thus affecting the parallel decomposition.

Privatization of the output structure is performed transparently to prevent data collisions. If the `out` structure is parametrized by the iterated dimension, then the different workers access different places in the memory, and no privatization is necessary. Otherwise, the algorithm creates a local copy of the `out` structure for each worker thread as needed (managed by `tbb::combinable`), allocating appropriate memory when the given copy is used for the first time. The copies are merged at the end using the third lambda.

4.1 Extension to GPU (CUDA traverser)

One of the key advantages of the proposed abstraction is that it aims at maximal compatibility with standard C++ compilers. This simplifies and expedites its application within other parallel environments like CUDA, which employs its custom compiler that adds some extensions but remains compatible with C++ language. We present an adaptor that allows applying traversers for kernel execution and one particular construct that becomes especially useful when privatizing data structures in shared memory.

CUDA framework is based on the data-parallel paradigm and uses thread abstraction to achieve parallelism. CUDA threads are spawned collectively (forming a *grid*) executing a single piece of code (*kernel*). Each thread is given index structures (`threadIdx`, `blockIdx`), which identify a data element to be processed by the thread. Additionally, threads are grouped into *thread blocks* so they can cooperate more

[2]https://github.com/jiriklepl/PMAM2024-artifact

closely (e.g., via *shared memory* or using faster synchronization primitives). The indexing structures (for threads and blocks) can encompass up to three dimensions, so the model is more convenient for programmers when dealing with multidimensional data (like matrices or 3D grids).

From the perspective of traversers, the CUDA grid is mapped to selected loop dimensions. The original traverser can be transformed to achieve the desired mapping — i.e., which parts of the traversal are executed (possibly) concurrently and which are handled inside a CUDA thread. The following code represents a kernel that computes the histogram (stored in global memory) using atomic updates (a typical implementation) and where each thread computes multiple input values. The aggregation of work per thread is one of the common optimizations. In this case, it could produce more coalesced loads from global memory and prepare grounds for more elaborate optimizations like shared memory privatization, which we discuss further in the paper.

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

`in_trav` is an inner traverser created from the traverser of the input data in the kernel invocation (see below), and it covers the data traversed by a single thread. The invocation is handled as follows.

```
auto in_blk_struct = in_struct
    ^ noarr::into_blocks<'i', 'B', 't'>(BLOCK_SIZE)
    ^ noarr::into_blocks<'B', 'b', 'x'>(ELEMS_PER_THREAD);
auto in = noarr::make_bag(in_blk_struct, in_ptr);
auto out = noarr::make_bag(out_struct, out_ptr);

auto ct = noarr::cuda_threads<'b', 't'>(noarr::traverser(in));
histogram<<<ct.grid_dim(), ct.block_dim()>>>(ct.inner(), in, out);
```

The essential part of the mechanism is hidden in the function `cuda_threads` that automatically associates the dimensions of the traverser with the dimensions of the CUDA grid — in this case, letting the `b` be the index of the block and `t` the index of the thread within the block. The resulting *cuda traverser* is then used to provide kernel invocation parameters by `grid_dim()` and `block_dim()` calls and infer the inner traverser that is passed as an argument of the kernel. The inner traverser binds its `b` and `t` dimensions to the `blockIdx` and `threadIdx` structures respectively, and allows (in-thread) iteration over the remaining dimension `x`.

Let us emphasize that the execution, as well as internal behavior (how many items are processed by a thread), are both governed by the traverser. That permits a certain level of agnosticism in the parallelization of algorithms. Furthermore, the composable nature of traversers makes it possible to separate the blocking operations required for CUDA execution to be prepared in a separate structure applied by `order()` method. Furthermore, since the kernel invocation is a common operation, Noarr also provides a method `simple_run()`, which can be used instead as a shortcut.

4.2 Shared memory privatization

Massively parallel systems are particularly susceptible to intensive data synchronization. In the `histogram` kernel presented in the previous section, the many simultaneous atomic updates cause a bottleneck. Even if the updates are distributed evenly, collisions are unavoidable since the histogram has much fewer bins than the GPU has cores.

A typical solution to this problem is data structure privatization — i.e., creating multiple copies of the histogram so each thread (or a small group of threads) has a separate copy. In this case, the optimal solution is to create a copy for each warp lane (32 copies per thread block) and place it in the shared memory. This way, threads running in lockstep have no collisions among themselves. The result aggregation in the shared memory significantly decreases the number of global memory accesses. Then, the individual copies need to be merged into the final copy in the global memory before a thread block concludes its execution.

The shared memory has a specific hardware design — it is divided into 32 independent memory banks (consecutive 32-bit words are placed in banks in a round-robin fashion), so each thread in the warp can access a different bank. Concurrent operations accessing one bank are serialized (except for special cases like data broadcast), which delays an entire warp. Histogram stored in a contiguous block in the shared memory would span over all banks, so concurrent updates would still cause bank conflicts (and thread serialization) even if the structure is privatized. The solution is to place each histogram copy into a separate bank, which requires a rather specific stridden layout pattern.

We introduce `noarr::cuda_striped<N>`, a helper structure tailored particularly for shared memory. The parameter N denotes the number of copies distributed across the banks. The optimum is $N = 32$ (i.e., one copy per bank); however, picking a lower N may be necessary if 32 copies would not fit in the memory. The kernel could be optimized using a striped structure `shm_s`, as follows. (For the sake of brevity, we omit initialization, reduction, and the necessary barriers.)

```
template<class InT, class I, class ShmS, class O>
__global__ void histogram(InT in_trav, I in, ShmS shm_s, O out) {
    extern __shared__ char shm_ptr[];
    auto shm_bag = make_bag(shm_s, shm_ptr);
    // initialize shared memory (zero the bins)
    in_trav.for_each([=](auto state) {
      auto val = in[state];
      atomicAdd(&shm_bag[noarr::idx<'v'>(val)],1);
    });
    // reduce shm copies and write the histogram in global memory
}
```

Note that the `atomicAdd` merely uses the bag allocated in the shared memory, and the `shm_s` structure transparently ensures the appropriate privatized copy is accessed (based on the `threadIdx` value). The construction of `shm_s` structure is performed externally (as well as the shared memory allocation) in our example, so the kernel is more generic, and the shared memory utilization can be subjected to external tuning; however, if required, it can be constructed internally.

```
// 'in' and 'out' match the previous example
auto ct = noarr::cuda_threads<'b', 't'>(noarr::traverser(in));
auto shm_s = out_struct ^ noarr::cuda_striped<NUM_COPIES>();
histogram<<<ct.grid_dim(), ct.block_dim(),
    shm_s | noarr::get_size()>>>
    (ct.inner(), in, shm_s, out);
```

The shared memory needs to be initialized when each thread block starts — in this particular case, all histogram copies need to have their bin counters zeroed. The most efficient way is for all threads (of a block) to cooperate on initialization evenly. For this purpose, we use `noarr::cuda_step`, which automatically distributes the work among the available threads. The `cuda_step` object is constructed using the rank of the current thread and the number of threads cooperating on the stripe provided by `current_stripe_cg`.

```
auto subset = noarr::cuda_step(shm_s.current_stripe_cg());
noarr::traverser(shm_bag).order(subset).for_each(
    [=](auto state) { shm_bag[state] = 0; });
```

A different access pattern is required at the end, where the histogram copies are merged. In this case, the threads cooperatively iterate over the histogram, processing the bins concurrently. Each bin is summed up across the copies and atomically added to the global structure. The `num_stripes` method returns the number of copies. The difficulty here is that we cannot access the shared memory bag directly since it would direct each thread to its corresponding copy, so the actual index (state) needs to be computed as follows.

```
noarr::traverser(out).order(noarr::cuda_step_block())
  .for_each([=](auto state) {
    size_t sum = 0;
    for (size_t i = 0; i < shm_s.num_stripes(); ++i) {
      sum +=
        shm_bag[state.template with<noarr::cuda_stripe_index>(i)];
    }
    atomicAdd(&out[state], sum);
  });
```

Granted, the code required to access all private copies from each thread is rather complex. However, this type of access is required only for the final reduction, and such an operation can be easily wrapped in a templated algorithm, so the regular user would not have to implement it explicitly.

5 Evaluation

The evaluation has two objectives: We would like to demonstrate that the proposed abstraction has no additional performance overhead, and we discuss its qualities from the perspective of the programmers using simple code metrics.

The most important results are presented in the remainder of this section. The complete set of experiments and results is available in the replication package.

5.1 Methodology and datasets

The presented experiments were measured on Intel Xeon Gold 6130 (CPU) and Tesla V100 PCIe 16GB (GPU) compiled with GCC 12.2 and NVCC 12.2. Each test comprised one warmup run and 10× subsequent measured runs on the `EXTRALARGE` dataset. The wall time of the tested kernel was measured by a high-resolution system clock. As expected, the variance of the measured times was very low (below 1%), so we present only the mean values.

We used *Polybench/C-4.2.1*[3] and *Polybench/GPU-1.0*[4] benchmark suites for the performance evaluation. Polybench/C-4.2.1 suite (CPU kernels) contains a set of 30 algorithms commonly used in scientific high-performance computing, such as problems from linear algebra, stencils, or data mining. The Polybench/GPU suite contains a set of 21 algorithms mostly from the Polybench/C suite, with the addition of some algorithms that are more specific to GPU computing (such as 2DConvolution). For Polybench/GPU, we have implemented 5 algorithms as a representative subset for the evaluation.

5.1.1 Threats to validity.

The greatest concern is whether our Noarr implementation is comparable with the original Polybench code. To mitigate this threat to validity, we have imposed several rules that govern the transcription of Polybench kernels into their Noarr counterparts:

1. All data layouts are equivalent; each dimension of a data structure is represented by `noarr::vector`.
2. The loops from the baseline implementation are directly mapped to equivalent Noarr iterative constructs (such as methods `for_each` and `for_dims`).
3. Kernels are structurally equivalent, and their computation statements are in the same order and rewritten into an equivalent form.
4. Accesses into data structures are at the equivalent computation points.
5. The time measurements and device synchronizations (for GPU) take place at equivalent program points.

Rewriting the algorithms according to these requirements is not easily automatable and it takes an extensive programming effort. However, as a sanity check, we have included scripts that check whether the implementations produce the same result.

5.2 Performance results

The performance results are presented as a relative speedup of Noarr implementations over their corresponding plain C/C++ (or CUDA) counterparts. Speedups above 1× indicate that the Noarr implementation enabled additional compiler optimizations, whereas speedups below 1× indicate possible overhead or that Noarr prevented some optimizations.

Figure 2 summarizes the results of the entire Polybench in sequential execution. Most of the algorithms indicate that Noarr implementation has the same performance as plain C. There are four outliers where Noarr performed better and four where it performed worse than the baseline. Examining the compiled code indicates that the differences are caused by the compiler selecting a different optimization path.

[3] https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1
[4] https://github.com/sgrauerg/polybenchGpu

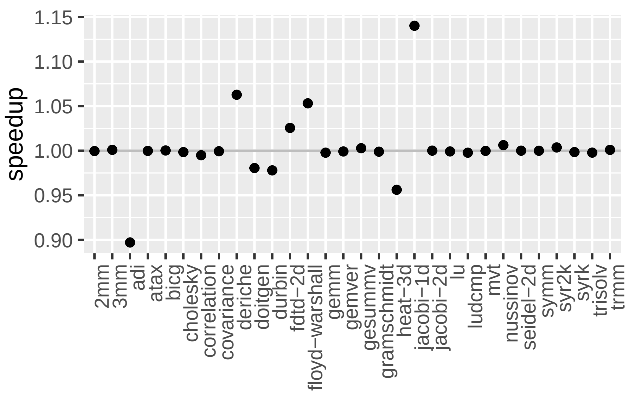

Figure 2. Comparing Noarr to plain C on Polybench/C-4.2.1

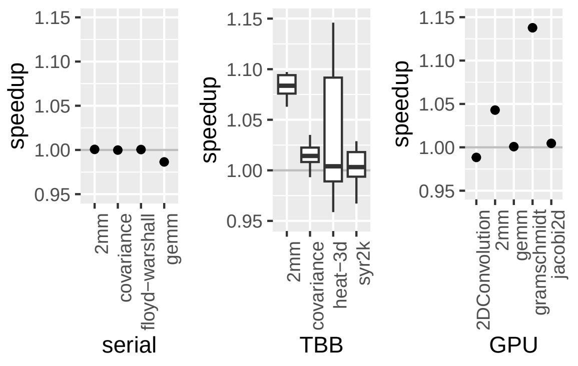

Figure 3. Comparing selected algorithms Noarr vs. plain C/C++/CUDA: tuned for performance (left), TBB parallelization (middle), GPU parallelization (right)

Figure 3 (left) presents the speedups of a selected subset of Polybench algorithms that were subjected to tuning (applying tiling and loop reordering). The middle graph presents the results of selected algorithms with their outermost loop in the critical segment parallelized using TBB. The GPU results (using CUDA traverser) are presented in the right graph of Figure 3. The results indicate that neither the additional traverser transformations applied in Noarr nor the parallelization extensions have any significant overhead over direct implementation in C, TBB, and CUDA, respectively. The parallel processing on a multi-socket CPU host is much more volatile, so we present the boxplots of all ten results instead of the mean value in the TBB graph.

5.3 Discussing code design aspects

Comparing the loop transformation approaches from the code design perspective is very challenging for many reasons. A user study might be the best way, but it is currently beyond our capabilities as it would require the cooperation of many users. For the basic insight, we provide a discussion comparing three typical approaches (annotations, DSL, and native C++ with the assistance of Noarr). Details about our selection of the compared technologies are in Section 6. We use the matrix multiplication running example optimized for memory transfers by blocking.

```
float A[I][K], B[K][J], C[I][J];

for (i = 0; i < I; i++)
    for (j = 0; j < J; j++)
        for (k = 0; k < K; k++)
Comp:       C[i][j] += A[i][k] * B[k][j];

affine(Comp, {[i,j,k]->[i,k,j]})
affine(Comp, {[i,j,k]->[i1,j1,k1,i2,j2,k2]: i1=[i/32] and i2=i%32
    and j1=[j/32] and j2=j%32 and k1=[k/32] and k2=k%32})
```

listing 1. Loopy (using `affine` compiler directives)

Listing 1 presents an implementation that relies on annotations. It keeps the code quite close to the original (plain C) implementation since the entire transformation is described by separate `affine` constructs. On the other hand, these constructs are quite complex to understand at first glance and limited to affine transformations only.

```
Halide::Buffer<float> A{I, K}, B{K, J}, C{I, J};

Halide::Func Comp{"Comp"};
Halide::Var i{"i"}, j{"j"};
Halide::RDom k{0, K};

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

Halide::Var i2{"i_inner"}, j2{"j_inner"};
Halide::RVar k1{"k_outer"}, k2{"k_inner"};

Comp.update().reorder(i, k, j)
    .tile(i, j, i2, j2, 32, 32).split(k, k1, k2, 32);

Comp.realize(C);
```

listing 2. Halide (DSL using methods on function stages)

The Halide implementation (Listing 2) represents the DSL approach. Halide was designed for regular operations like matrix multiplication; thus, the realization is easy, albeit a little more verbose than Loopy and Noarr. On the other hand, with more complex data dependencies or irregular data traversals (for instance, the Gram-Schmidt algorithm from Polybench), Halide implementation gets quite cumbersome.

```
auto A = bag(scalar<float>() ^ array<'k', K>() ^ array<'i', I>());
auto B = bag(scalar<float>() ^ array<'j', J>() ^ array<'k', K>());
auto C = bag(scalar<float>() ^ array<'j', J>() ^ array<'i', I>());

auto my_order = into_blocks<'i', 'I', 'x'>(32) ^
                into_blocks<'j', 'J', 'y'>(32) ^
                into_blocks<'k', 'K', 'z'>(32) ^
                reorder<'I', 'K', 'J', 'x', 'z', 'y'>();

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

listing 3. Native C++ with Noarr traversers

Finally, Listing 3 presents our implementation in Noarr. The complexity is comparable both with Loopy and Halide, though the assembling of structures and traverser ordering may seem a little unusual for mainstream C++ programmers since it uses functional programming patterns. The greatest benefit is that the type constructs for structures and orderings can be easily reused, which simplifies the design of similar data structures and the optimization of similar algorithms. Furthermore, this code can be compiled by any C++ standard-compliant compiler without extra preprocessing.

To assess the implementation overhead of Noarr compared to simple C code, we extracted the corresponding kernel codes delimited by the `scop` pragmas and formatted them using `clang-format`. We then compared them using coding metrics. On average, a Noarr implementation contains 11.28% more lines of code and 31.95% more individual code tokens than the baseline implementation. When compressing each kernel with gzip, the average Noarr implementation is 41.19% larger than the C baseline. This figure drops to 28.67% when comparing the gzipped tar archives containing all kernels. These results demonstrate that direct reimplementation using the proposed abstraction increases the size of the source code by approximately a third. However, the added code agnosticism makes the proposed approach superior when there is a need for at least two versions of the same algorithm (eliminating the need for code duplication) or when frequent modifications in the traversal order are required (handled by updating just the transformation structure).

6 Related Work

Optimization based on loop transformations has been addressed from various perspectives in vast research materials, namely in the fields of compilers, vectorization, autotuning, code generators, and optimizations of particular scientific computations. Contemporary compilers use sophisticated loop optimizers based on the polyhedral model, such as Graphite in GCC [15] or Polly in LLVM [9]. However, these optimizers are limited by the lack of information about the effects of the transformations on the optimized metric.

One of the first papers [6] that addressed the loop transformations from the perspective of optimizing memory operations is over 20 years old. It proposed using Ehrhart polynomials to compute how many times a single index reference is computed in a loop. Since then, several models based on static predictions have been created [9, 15]. The most recent innovations focus on elaborate multi-objective scheduling for loop transforms [4].

Autotuning methods have also addressed this problem by generating various variations of the tuned program and evaluating them either by sophisticated models or by measuring execution metrics such as execution time. Modern autotuning tools are often built on top of existing optimizers and employ methods from the machine-learning domain — for instance, Wu et al. [17] presented a tuning tool based on Polly [9] that employs Bayesian optimizations.

6.1 Domain Specific Languages

Many works address the issue of separating the specifics of memory access patterns and traversals from the algorithm itself by defining the algorithm via some DSL with a simplified model that facilitates applying various transformations. We have selected two representatives used in state-of-the-art production code. Our approach can be superficially related to theirs, with the fundamental distinction of their approach relying on a custom compilation pipeline and a runtime library, while Noarr is compiled by standard C++ compilers.

The Halide language [13] follows a decoupling approach similar to our combination of traversers and proto-structures. Halide primarily focuses on image processing, but their approach found use even for optimizing deep learning algorithms, as shown by the work of Apache TVM [5]. In their approach, the definition of an algorithm is followed by a *schedule* that represents various traversal transformations. The schedules roughly correspond to our idea of traversers and their transformations via proto-structures, but they lack any support for more complex or user-defined traversals (such as the z-curve).

6.2 Annotations

Another approach employed by various tools and compiler extensions uses code annotations that suggest the desired way of handling data structure layouts or loop transformations. Most of the tools rely on pragma directives, but the annotations can also be provided in comments, as regular statements and expressions [11], or via XML syntax [18].

Loopy [11] is a system for loop transformations designed as an extension to the LLVM compiler, which is perhaps the closest to our research since it relies on programmer-guided loop transformations. Building upon a polyhedral compilation library, it provides custom affine transformations with the addition of testing for the legality of loop transformation.

6.3 Native tools

The projects closest to our approach can be characterized by being built using abstractions provided by the C++ language itself and thus allowing for more seamless interaction with other C++ features, various intrinsics, or user-defined abstractions. This also avoids the necessity for custom development toolkits in favor of existing tools for C++ development, greatly reducing requirements on maintenance.

The C++ Standard Library already provides an abstraction for different traversal options via its ranges library. However, the library is not designed for parallelism and does not support multidimensional data layouts. The most common layouts of multidimensional arrays are expressible via the `mdspan` class template, but this abstraction lacks generality and does not provide a way of expressing traversals.

The NVIDIA Thrust library [3] provides routines for parallel code execution on both CPU and GPU. It is a template library based on the C++ Standard Library. While providing plenty of freedom in defining systems-agnostic concurrent traversals via functions like `thrust::for_each` or `thrust::reduce`, their approach is based on an iterator design pattern restricted to 1D traversals. Furthermore, Thrust is restrained to rather high-level use cases by not exposing low-level CUDA API (such as thread or block index).

Similarly to Thrust, Kokkos [16] and RAJA [1] provide routines for common parallel idioms (`for_each`, `reduce`, `scan`) and they serve as portability layers for many systems such as HIP, OpenMP, CUDA or SYCL. However, they primarily focus on platform-agnosticism and do not provide the necessary abstractions for expressing traversal transformations.

7 Conclusion

We have presented a novel object-oriented approach for user-guided loop transformations focusing on the traversal of regular data structures. We base the abstraction on the Noarr library, expanding the Noarr paradigm for layout design to encompass loop transformations. This expansion significantly enhances the versatility of Noarr, enabling users to optimize memory access patterns by altering either the data structure layout, the traversal pattern, or both — all via a unified mechanism of applying composable first-class transformation objects. This approach promotes code independence (emphasizing separation of concerns) and reusability. It also simplifies semi-automated experimentation and performance tuning. Building the abstraction on top of Noarr (which automatically handles correct indexing and iteration ranges) further simplifies the transformation design process and makes it less prone to errors.

Besides the benefits related to memory access optimizations, the traverser abstraction is particularly useful for parallel processing. We demonstrate its utility with two implementation examples (TBB and CUDA) as proof of concept. Furthermore, we introduce an extension of Noarr that handles the management of replicated structures in CUDA shared memory. This functionality is particularly relevant in General-Purpose computing on Graphics Processing Units (GPGPU) programming.

Acknowledgments

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

References

[1] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. 2019. RAJA: Portable performance for large-scale scientific applications. In *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*. IEEE, 71–81.

[2] David Bednárek, Martin Kruliš, and Jakub Yaghob. 2021. Letting future programmers experience performance-related tasks. *J. Parallel and Distrib. Comput.* 155 (2021), 74–86.

[3] Nathan Bell and Jared Hoberock. 2012. Thrust: A productivity-oriented library for CUDA. In *GPU computing gems Jade edition*. Elsevier, 359–371.

[4] Lorenzo Chelini, Tobias Gysi, Tobias Grosser, Martin Kong, and Henk Corporaal. 2020. Automatic generation of multi-objective polyhedral compiler transformations. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 83–96.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799* 11, 20 (2018).

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

[7] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[8] Zhangxiaowen Gong, Zhi Chen, Justin Szaday, David Wong, Zehra Sura, Neftali Watkinson, Saeed Maleki, David Padua, Alexander Veidenbaum, Alexandru Nicolau, et al. 2018. An empirical study of the effect of source-level loop transformations on compiler stability. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29.

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

[10] Junjie Li, Sanjay Ranka, and Sartaj Sahni. 2013. GPU matrix multiplication. *Multicore Computing: Algorithms, Architectures, and Applications* 345 (2013).

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

[12] Chuck Pheatt. 2008. Intel® threading building blocks. *Journal of Computing Sciences in Colleges* 23, 4 (2008), 298–298.

[13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[14] Adam Šmelko, Martin Kruliš, Miroslav Kratochvíl, Jiří Klepl, Jiří Mayer, and Petr Šimůnek. 2023. Astute Approach to Handling Memory Layouts of Regular Data Structures. In *Algorithms and Architectures for Parallel Processing: 22nd International Conference, ICA3PP 2022, Copenhagen, Denmark, October 10–12, 2022, Proceedings*. Springer, 507–528.

[15] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. 2010. Graphite two years after: First lessons learned from real-world polyhedral compilation. In *GCC Research Opportunities Workshop (GROW'10)*.

[16] Christian R Trott, Damien Lebrun-Grandie, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S Hollman, Dan Ibanez, et al. 2021. Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 805–817.

[17] Xingfu Wu, Michael Kruse, Prasanna Balaprakash, Hal Finkel, Paul Hovland, Valerie Taylor, and Mary Hall. 2022. Autotuning PolyBench Benchmarks with LLVM Clang/Polly loop optimization pragmas using Bayesian optimization. *Concurrency and Computation: Practice and Experience* 34, 20 (2022), e6683.

[18] Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan. 2007. POET: Parameterized optimizations for empirical tuning. In

2007 IEEE International Parallel and Distributed Processing Symposium. IEEE, 1–8.