Data Layout Optimization in RAJA

ABSTRACT

The layout of data in memory is a key consideration in high performance computing applications. From reducing cache and page misses to relieving pressure on memory bandwidth and avoiding inter-process communication, using a good data layout improves performance at all levels of an application. RAJA, a C++ performance portability library, incorporates the layout of an application's data into its interface Thus, a developer can try different layouts without major refactoring costs. However, some applications benefit from changing the data layout mid-computation to facilitate better locality. Implementing such a mid-computation layout change in RAJA can improve performance, but is laborious to implement and lacks portability. This work remedies RAJA's shortcoming by introducing a lightweight, declarative API for changing data layouts between computations. Also, we present a performance model for layouts using the loop bounds and data access order as input and a runtime interface that dynamically estimates performance to select an appropriate layout. These systems combine so that the model and runtime "fill in" layout choices that the user did not specify. Thus the user can specify as much of the layout information as they please and still obtain the performance benefits of changing data layouts. Further, our system is built directly into the RAJA library, meaning that no additional build steps are required. We evaluate our system on four benchmarks, where it achieves similar performance improvements to hand-implemented optimizations.

KEYWORDS

data layout, data optimization, loop chains, inter-loop optimization, performance model

ACM Reference Format:

. 2022. Data Layout Optimization in RAJA. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

How data is stored and accessed has always been recognized to have a significant impact on performance. For example, on a typical cache-based system, iterating through a two-dimensional array in column-order when data is stored in row-order can lead to significant slowdown Although mechanisms such as compiler optimizations (loop permutation/interchange) and data layout policies like those in Kokkos [5] and RAJA [9] can align data layout with computation schedules, they become difficult to automate and laborious to manage by hand as the number of loops and arrays grow.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

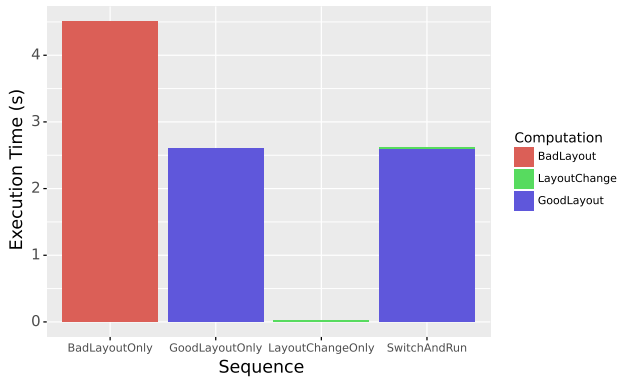


Figure 1: Execution time for matrix multiplication using optimal and non-optimal data layouts and execution time for converting from non-optimal to optimal.

In this paper, we present an approach for exposing data layout decisions to the programmer in the C++ portable, parallel library RAJA. The presented API allows the programmer to specify as many data layout decisions in a sequence of data-sharing loops (otherwise known as a loop chain [12]) as they choose. Then, using a portable cost model based on microbenchmarking, our system identifies any remaining layout changes estimated to improve performance.

Figure 1 shows the importance of a good data layout. The BadLayout column shows the execution time for a matrix multiplication where the three arrays are laid out opposite of their access order. The GoodLayout column shows the execution time for the same matrix multiplication when the arrays are laid out to match their access order. The LayoutChange column shows the execution time of code that changes the layout from the bad layout to the good one. Finally, the SwitchAndRun column shows the combined execution time of changing the layout from bad to good and then running. As the graph shows, performance improves significantly (72%) simply by changing the data layout, and the cost of performing the layout change is small (<1% of execution time).

Although controlling data layout is quite important, the options programmers have to control data layouts in coordination with loop schedules have limitations. Compiler optimizations such as loop permutation have been developed to change loop schedules so that a loop traversal order better aligns with data layout. However, general purpose compilers (1) have difficulties analyzing for the relationship between data layout and schedule due to challenges like aliasing [7] and lack of multidimensional arrays in C/C++ languages; (2) do not generally expose fine-grained data layout or loop optimization controls to users, leading to proposals to add pragma-based controls to OpenMP [15] and Clang [14]; and (3) typically do not optimize across multiple loops well.

The problem of inter-loop optimizations, especially those that balance parallelism and locality, has spawned many approaches. Approaches are often built into the compiler, lacking a user-accessible interface. Some use loop schedule transformations alone [17, 24],

while others consider data layout transformations as well [1–3, 10, 11, 20]. Other approaches develop or expand domain specific languages to expose controls to the user. Many such approaches target stencil codes [6, 13, 16] and image processing [18, 22].

General purpose libraries like Kokkos and RAJA and programming languages like Chapel [4] give programmers some control over the data layout of multi-dimensional arrays. However, this support is limited to the point of initialization. Thus, programmers must use that particular data format for the entire computation. While a developer could change the data layout mid-computation in RAJA, the developer must either modify RAJA library code or use multiple arrays for the same data. Both options increase the code complexity and increase its fragility. Even for the brave developer choosing one of these options, yet another obstacle remains: selecting the right combination of formats for the computation. Even a modest computation of two loops using four 2D arrays has more than 200 combinations from which to choose, meaning trying out all possible options quickly becomes infeasible.

We extend the RAJA C++ library to address this gap in support for data layout optimizations. First, our extension exposes a lightweight API for implementing mid-computation data layout optimizations. For those with specific format changes in mind, two methods — `set_format_before` and `set_format_after` — ensure the appropriate format is used[1]. While the user can specify as many format requirements as they desire, any left unspecified are completed using a performance model. Guided by run-time microbenchmarking, the model selects format changes based on the solution of an Integer Linear Programming (ILP) problem that considers the cost of converting formats against the potential improvement with an alternative format. Our inclusion of a performance model means a developer can obtain performance improvement without having to determine the sequence of format changes themselves or use an autotuner to try every possible option.

This paper includes the following contributions:

- An API in RAJA for specifying data layouts and data layout transformations;
- A performance model to select data layouts based on run-time microbenchmarking and a data layout transformation strategy for a loop chain;
- An optimization that increases the reuse of microbenchmarking information; and
- Performance results and source lines of code (SLOC) for five variants of four benchmark kernels comparing hand-implemented optimizations to our system.

2 RAJA AND RAJALC

We use a variety of components of RAJA to enable automatic data format transformations, as well as features of the loop chain extension RAJALC [19]. This section reviews how a computation is described in RAJA, working with the implementation of the 3MM kernel found in Listing 1.

The foundational elements of RAJA are the execution constructs, `forall` and `kernel`. These templated functions execute a loop immediately, whereas the RAJALC `make_forall` and `make_kernel` extensions create wrapper objects that are executed through the call operator. Calls to `make_kernel` can be seen in lines 30 through 32 in Listing 1. While the RAJALC functions create computation objects rather than immediately executing the computation, their interfaces are the same as their base RAJA counterparts.

[1]Since the control is completely exposed, an autotuner can also try out different options.

```
1  auto layout_01 = make_permuted_layout(sizes, {{0,1}});
2  auto layout_10 = make_permuted_layout(sizes, {{1,0}});
3  View2D A(A_data, layout_01);
4  View2D B(B_data, layout_10);
5  ... initializations proceed similarly for C-G with layout_01
6
7  auto loop_body1 = [&](auto i0, auto i1, auto i2) {
8    E(i0, i1) += A(i0,i2) * B(i2,i1);
9  };
10 auto loop_body2 = [&](auto i0, auto i1, auto i2) {
11   F(i0, i1) += C(i0,i2) * D(i2,i1);
12 };
13 auto loop_body3 = [&](auto i0, auto i1, auto i2) {
14   G(i0, i1) += E(i0,i2) * F(i2,i1);
15 };
16
17 using POLICY = KernelPolicy<
18   statement::For<0,omp_parallel_for_exec,
19     statement::For<1,loop_exec,
20       statement::For<2,loop_exec,
21         statement::Lambda<0>
22       >
23     >
24   >
25 >;
26
27 auto one_seg = RangeSegment(0,N);
28 auto segs = make_tuple(one_seg, one_seg, one_seg);
29
30 auto knl1 = make_kernel<POLICY>(segs, loop_body1);
31 auto knl2 = make_kernel<POLICY>(segs, loop_body2);
32 auto knl3 = make_kernel<POLICY>(segs, loop_body3);
33
34 knl1();
35 knl2();
36 knl3();
```

Listing 1: The running example for this paper. We start with an implementation of the 3MM kernel (E = A * B, F = C * D, G = E * F) in RAJA.

The execution constructs separate the specification of the computation from the specification of its schedule. The template parameter describes the schedule of the computation as an execution policy, defined on lines 17 through 25. Each level of the loop has its own schedule, where `omp_parallel_for_exec` indicates an OpenMP parallel loop and `loop_exec` indicates the compiler should make the decision. Other policies exist for vectorization and GPU-offloading. The runtime parameters describe the computation itself. The first parameter is the iteration space for the loop, defined on lines 27 and 28. The second parameter is the loop body to execute for each iteration, passed as a lambda. The three loop bodies are defined on lines 7 through 15. After the computation objects are created on lines 30 through 32, they are executed through the call operator on lines 34 through 36.

RAJA also provides an array wrapper class called a View. The View object has a number of capabilities that make it a valuable tool within RAJA codes. First, Views use the call operator to perform memory accesses. By overloading this call operator for symbolic iterator types, RAJALC enabled the runtime symbolic evaluation of kernels that use Views. RAJALC used the access information

gathered from symbolic evaluation to ensure the correctness of its scheduling optimizations. This work uses RAJALC's runtime symbolic evaluation to inform our performance model.

Second, Views fully parameterize their underlying data formats with the Layout object. Consider a programmer who wants to switch their data from row-major to column-major. Without Views, every access to their data `A[i][j]` has to be changed to `A[j][i]` *and* the definition of the array needs to be changed. This is prohibitively expensive, especially when the programmer does not yet know the performance impact of such a decision. With Views however, the only change the programmer needs to make is to the View's definition: the layout permutation changes from $(0, 1)$ to $(1, 0)$. In Listing 1, `A` is defined with the normal $(0, 1)$ layout, but `B` is defined with the $(1, 0)$ layout. Note that the change in `B`'s layout does not change how it is used in the computation.

3 USER SPECIFICATION OF DATA FORMAT

Throughout a computation, different parts of the computation access data in different orders. For example, consider the Views `A`, `B`, and `D` in Listing 1. The order in which `A` is accessed is different from `D` because the argument order in their accesses are different. In contrast, while `B` and `D` have the same argument order, their access order is still different because they have different layouts. Looking at the two references to `F` in kernels two and three, we can see that even access order to the same data can change through a computation. Because different formats are optimal for different kernels, this creates an opportunity for optimization.

However, RAJA's built-in support for changing data layouts is minimal. While Views can be instantiated with different layouts, changing the layout of an existing View is not as simple. This is because changing layouts also requires reordering the underlying data to match the new layout. To implement such a layout change by hand requires the programmer to allocate a new temporary array, copy the data from the View to the temporary array in the right order, copy the data *back* to the memory in the View, and then finally update the View's layout object.

This work removes that barrier by expanding the declarative data optimization system begun by RAJALC. While RAJALC tackled the problem of scheduling optimizations, we target making data format changes between computations. The new `FormatDecisions` object is the central component of our system. Its instantiation takes a tuple of references to Views that are possible targets of format changes and the kernel objects that constitute the whole computation. Two methods are used to register desired formats: `set_format_before` and `set_format_after`. Both take the View to be reformatted, the desired format, and the computation before or after which the desired format should be used. Once all format choices are registered, the complete computation with the desired format conversions is generated using the `finalize` method.

4 PERFORMANCE MODELING

In addition to the format changes registered by the user, the new `FormatDecisions` also uses a performance model to determine if additional format changes will improve performance. This capability means that even without any registered format choices, `FormatDecisions` often selects the same changes the user would

```
auto knl1 = make_kernel<KPOL>(segs1, [=](auto i0, auto i1, auto i2) {
  E(i0, i1) += A(i0, i2) * B(i2, i1);
});
auto knl2 = make_kernel<KPOL>(segs2, [=](auto i0, auto i1, auto i2) {
  F(i0, i1) += C(i0, i2) * D(i2, i1);
});
auto knl3 = make_kernel<KPOL>(segs3, [=](auto i0, auto i1, auto i2) {
  G(i0, i1) += E(i0, i2) * F(i2, i1);
});

auto decisions = format_decisions(tie(B,D,F), knl1, knl2, knl3);

decisions.set_format_before(B, {{1,0}}, knl1);
decisions.set_format_before(D, {{1,0}}, knl2);

decisions.set_format_before(F, {{0,1}}, knl1);
decisions.set_format_after(F, {{1,0}}, knl2);

auto computation = decisions.finalize();
computation();
```

Listing 2: The 3MM benchmark implemented using FormatDecisions.

themselves choose. We encode the problem as an integer linear program where the solution represents our model's pick for the optimal layouts.

We use binary decision variables representing whether or not a particular format is used at different points in the chain. For example, there are eight decision variables for the `B` View in Listing 2, one for each of the two possible formats at each of the four points in the chain. While there are only three kernels in the chain, there is an additional point added for the "output" format that the View has after the computation is done. We also add constant variables for the format when the computation begins. We also use decision variables to represent the required format conversions. We construct separate models for each View in the computation.

Symbolically, we represent the variables as follows. To start, let F, K, and T represent the set of all formats, kernels, and conversion times. Because we add additional elements to K for the input and output formats, we know that $|K|$ is the number of user kernels plus 2. Furthermore, because there is one more conversion than kernel, we know that $|T| = |K| + 1$. For each kernel, we have one variable for each format. Thus, let $fmt_{f,k}$ denote the variable for using the format f during kernel k, where $f \in F$ and $k \in K$. For conversion variables, we have one variable for each possible conversion at each time point. Thus, let $conv_{i,o,t}$ denote the decision variable for converting from format i to to format o at conversion time t, where $i, o \in F$ and $t \in T$.

Four types of constraints are imposed on the decision variables.

- Format Uniqueness: At each time point, the View has exactly one selected format. Given by:
$$\bigwedge_{k \in K} (1 = \sum_{f \in F} fmt_{f,k})$$
- Conversion Uniqueness: At each conversion point, the View goes through exactly one conversion. Given by:
$$\bigwedge_{t \in T} (1 = \sum_{i \in F} \sum_{o \in F} conv_{i,o,t})$$
- Format-Conversion Matching, Input: At each conversion point, the input format for the conversion matches the

format of the preceding kernel. Given by:

$$\bigwedge_{t \in T} \bigwedge_{i \in F} fmt_{i,t.prev} = \sum_{o \in F} conv_{i,o,t}$$

- Format-Conversion Matching, Output: At each conversion point, the output format for the conversion matches the format of the following kernel. Given by:

$$\bigwedge_{t \in T} \bigwedge_{o \in F} fmt_{o,t.next} = \sum_{i \in F} conv_{i,o,t}$$

- User Prescription: All user-provided format choices are met. With all user choices U represented as pairs of kernels and the format the user wants for that kernel, given by:

$$\bigwedge_{(k,f) \in U} fmt_{f,k} = 1$$

Figure 2 shows the variables in a model for a 2D View for a computation with three kernels. For each kernel, we see two decision variables, one for using column-major and one for using row-major. Similarly, for each conversion point, we see four decision variables, one for each combination of input and output format. The green boxes indicate the terms that are part of a Format Uniqueness constraint and a Conversion Uniqueness constraint. The red and blue underline shows the part of the format and conversion decision variables that group them in the Conversion-Format Matching constraints.

The objective function is constructed using the estimated cost of the format choices and conversions. Each decision variable is assigned a cost coefficient in the following manner. First, we execute and time a small loop with similar access patterns to the choice, then cache the result for later decision variables. Second, we multiply the result by the number of iterations the choice affects. For example, a conversion decision for `B` in Listing 2 would be multiplied by the dimensions of `B`. In contrast, a format decision for `B` for `knl1` would be multiplied by all three loop dimensions.

While we can estimate the cost of format conversions without information about the kernels themselves, the cost of using a format for a kernel depends on how that kernel accesses the data. Thus, we need to gather information about how each kernel accesses the data that may be transformed. To do so, we use RAJALC's symbolic evaluation capabilities. By defining an overloaded call operator in the View class, we can gather access information for each kernel at runtime. Then, this access information is used to estimate the cost of using different formats. Once all cost estimates are added to the objective function, we use the Integer Set Library [23] to find the points in the space with the lowest overall cost estimates. Listing 3 shows a pseudocode algorithm for generating the objective function. Note that the costs associated with format variables are based on the uses of the Views within the computation. Thus, if one kernel makes multiple accesses to the same View, it will have one cost term for each access.

5 COST ESTIMATION

Because our system uses microbenchmarking to estimate the cost of different layouts, the overhead can meaningfully impact overall performance. To mitigate this effect, we reuse the microbenchmarking results as much as possible. To maximize reuse, we model choices

```
add_objective_function(view):
  obj_func = new ObjectiveFunction()

  # cost estimates for conversions
  for conversion_variable in conversion_variables:
    coefficient = conversion_variable.estimate_cost() * view.size()
    obj_func += (coefficient * conversion_variable)

  # cost estimates for accesses using different formats
  for kernel in kernels:
    view_accesses = kernel.evaluate_symbolically()
    for access to view in accesses:
      for fmt in formats:
        format_variable = get_format_variable(view, fmt, kernel)
        estimated_cost = format_variable.estimate_cost(access)
        coefficient = estimated_cost * kernel.size()
        obj_func += (coefficient * format_variable)
  set_objective_function(obj_func)
```

Listing 3: Algorithm for constructing objective function.

based on their access order, which describes the order in which iterations access data in memory.

By using the access order instead of the full description of the access, references to different Views can be modeled with the same estimates, even if they have different argument or policy orders. We argue that the access order is an accurate predictive metric for the relative performance of a layout choice.

5.1 Identifying an Access

The access pattern of any View reference is identified by its access order, which is in turn defined by its argument order, the kernel policy order, and the layout order. The argument order describes the order of the loop iterators in the reference with respect to the parameters of the lambda. In Listing 4, the references to `A`, `B`, and `C` have argument orders $(0, 2)$, $(2, 1)$, and $(0, 1)$, respectively. The kernel policy order describes the order in which the iterators are increased. This is equivalent to the nesting order of a loop. In Listing 4, references made by `knl1` have policy order $(0, 1, 2)$, while those may by `knl2` have policy order $(2, 0, 1)$. The indices of the `For` statements within the policy determine this order. The layout order describes the layout of data in memory. In Listing 4, `A` has layout order $(0, 1)$, `B` has $(1, 0)$, and `C` has $(0, 1)$. While each access has a layout order defined by the original layout of the data, during our modeling phase, different layout orders are estimated to see which is most effective for the particular kernel.

The access order describes the order in which the iterations of a kernel access the data in memory. For example, an access order of $(0, 2)$ indicates that the stride-one dimension of the data is traversed by the innermost (depth 2) loop and the larger stride dimension is traversed by the outermost (depth 0) loop. Similarly, an order of $(2, 1)$ indicates the stride-one dimension is traversed by the middle (depth 1) loop while the larger stride dimension is traversed by the innermost (depth 2) loop.

Calculating the access order of a View reference starts with the argument order. Then, we permute the argument order based on the layout order. Finally, we reorder the result based on the policy order. Using the reference to `B` in `knl2` as an example, we start with the argument order $(2, 1)$. The layout order of `B` is $(1, 0)$, so our intermediate result is $(1, 2)$. The policy order of `knl2` is $(2, 0, 1)$, so our final result is $(2, 0)$. This indicates that in `knl2`, the outermost

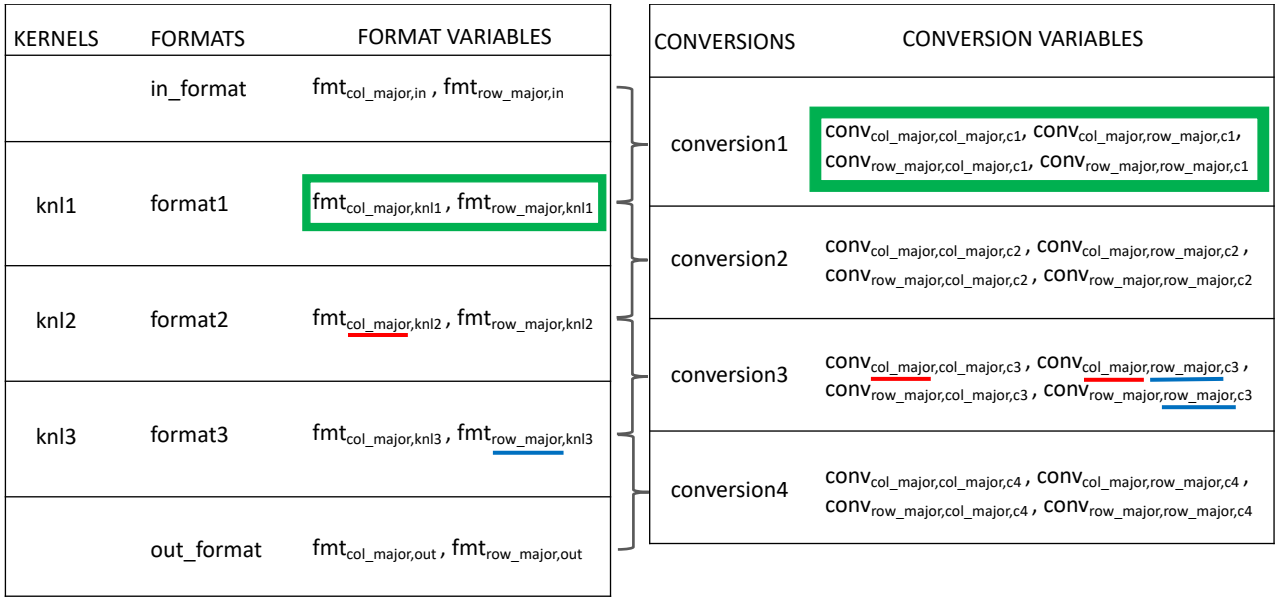


Figure 2: Formats, conversions, and decision variables for a 3 kernel computation. The green frames show example groups of variables that must sum to 1 due to the Format and Conversion Uniqueness Constraints. The red and blue underlines show examples of the Conversion-Format Matching constraints for input and output constraints respectively.

loop is indexing the stride-one dimension while the innermost loop is indexing the larger stride dimension.

Using the access order rather than the full description of the reference, we see that the reference to `B` in `knl1` will have similar performance impact to the reference to `C` in `knl2` For `B`, $(2, 1)$ becomes $(1, 2)$ becomes $(1, 2)$. For `C`, $(0, 1)$ becomes $(0, 1)$, becomes $(1, 2)$. This equivalence demonstrates how the monotonically increasing policy and layout orders are identity transformations.

5.2 Access Order as a Performance Metric

Our claim is that the access order of an access is an accurate predictive metric for the relative performance of a layout choice. We support this claim empirically using performance data from three microbenchmarks. For each microbenchmark, we record the 5-run average execution time of all combinations of policy, layout, and argument order. These execution times are then grouped by access order. If our claim is valid, then the execution times for each access order will cluster and the groups of execution times will not overlap. Microbenchmark 1 is an access to a 3-dimensional View in a 3-dimensional loop. Microbenchmark 2 is an access to a 2-dimensional View in a 3-dimensional loop, as in a matrix multiplication. Microbenchmark 3 is an access to a 3-dimensional View in a 4-dimensional loop for a set argument order. Table 1 summarizes them in more detail, including data sizes. The outermost loop is parallelized using OpenMP. All evaluation in this paper is performed on the same system. We used a single node with two 22-core IBM Power9 CPUs and 256GB of CPU memory.

Figure 3a shows the execution times for the different access orders for microbenchmark 1. Overall, the six possible access orders show good differentiation and clustering. The access order $(2, 0, 1)$

Microbenchmark Number	# View Dimensions	# Loop Dimensions	Size per Dimension
1	3	3	128
2	2	3	512
3	3	4	64

Table 1: Microbenchmark details.

does show two subclusters, but remain highly distinct from the other access order groupings. Additionally, access orders $(0, 1, 2)$ and $(1, 0, 2)$ show some overlap, likely because the bulk of the performance improvement comes from the innermost loop in a nest traversing the stride-one dimension of the data. We also hypothesize that the different between orders $(0, 2, 1)$ and $(2, 0, 1)$ is due to better data partitioning for the threads in the $(0, 2, 1)$ order. Figure 3b shows the execution times for the different access orders for microbenchmark 2. This microbenchmark also shows good cluster and differentiation. Like microbenchmark 1, the access orders $(0, 2)$ and $(1, 2)$ have similar performance, likely for the same reason. Figure 3c shows the execution times for the different access orders for microbenchmark 3. A similar pattern as the previous microbenchmarks emerges here: grouping based on the position of the innermost iterator.

6 EVALUATION

We present a number of experiments to evaluate our system. First, we examine performance and productivity using four benchmarks from the polybench suite [21]. Then, for one of the benchmarks, we perform an exhaustive evaluation of all layout choices to further explore the accuracy of our performance model. All performance

```
auto a_layout = make_permuted_layout(a_sizes, {{0,1}});
auto b_layout = make_permuted_layout(b_sizes, {{1,0}});
auto c_layout = make_permuted_layout(c_sizes, {{0,1}});

View2D A(a_data, a_layout);
View2D B(b_data, b_layout);
View2D C(c_data, c_layout);

auto matmul_lambda = [&](auto i0, auto i1, auto i2) {
  C(i0,i1) += A(i0,i2) * B(i2,i1);
}
auto segs = make_tuple(RangeSegment(0,iN),
  RangeSegment(0,jN),
  RangeSegment(0,kN));
using Policy_012 = KernelPolicy<
  statement::For<0,loop_exec,
    statement::For<1,loop_exec,
      statement::For<2,loop_exec,
        statement::Lambda<0>
      >
    >
  >
>;

using Policy_201 = KernelPolicy<
  statement::For<2,loop_exec,
    statement::For<0,loop_exec,
      statement::For<1,loop_exec,
        statement::Lambda<0>
      >
    >
  >
>;

auto knl1 = make_kernel<Policy_012>(segs, matmul_lambda);
auto knl2 = make_kernel<Policy_201>(segs, matmul_lambda);

knl1();
knl2();
```

Listing 4: Two implementations of matrix multiplication using different kernel policies.

results in this paper were collected on a single node with a 44-core IBM Power9 CPU and 256GB of CPU memory. All compilation used Clang version 13.0.0.

6.1 Experiment 1: Polybench

Our first experiment examined the performance and programmer productivity impacts of our contribution. We targeted four benchmarks from the Polybench suite: 2MM, 3MM, GEMVER, and MVT. While other benchmarks in the suite have the potential for data layout optimization, they present imperfect nesting and non-constant loop bounds, which are not currently supported by our implementation. For each benchmark, we implemented different variants and compared their relative speedup and the number of code changes necessary to go from the original implementation to the variant implementation. We used RAJAPerf [8] to run our experiments with default data sizes.

2MM solves the matrix expression $A * B * C$ using two matrix multiplications. First, it computes $A * B$ and stores the results in a temporary array. Then this temporary result is multiplied by C. 3MM solves a similar matrix expression $A * B * C * D$ using three matrix multiplications. It computes $A*B$, then $C*D$, then multiplies the results. GEMVER computes a vector multiplication and adds the result to a matrix. Finally, MVT computes a **m**atrix **v**ector product and **t**ranspose.

Variant Name	Kernel Objects	Layout Changes	Chosen By	Written Using
`RAJA_OpenMP`	No	No	-	-
`RAJALC`	Yes	No	-	-
`Hand_Layout`	Yes	Yes	User	No API
`Format_Decision`	Yes	Yes	User	API
`Model_Choice`	Yes	Yes	Model	API

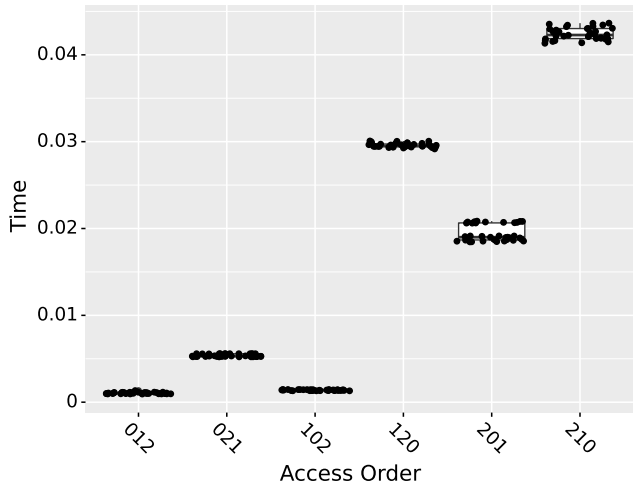
Table 2: Polybench variant descriptions.

Two of the four benchmarks, 2MM and 3MM, have higher loop dimensionality than data dimensionality. The remaining two, GEMVER and MVT, have the same loop and data dimensionality. This indicates that for GEMVER and MVT, the most profitable data transformation should be no transformation, as the traversal through the data for the conversion is the same traversal as that of the computation. For computations such as these, loop schedule transformations may be able to improve locality without the prohibitive cost of format conversions [10]. However, because of their higher loop dimensionality, 2MM and 3MM should benefit from format conversions.

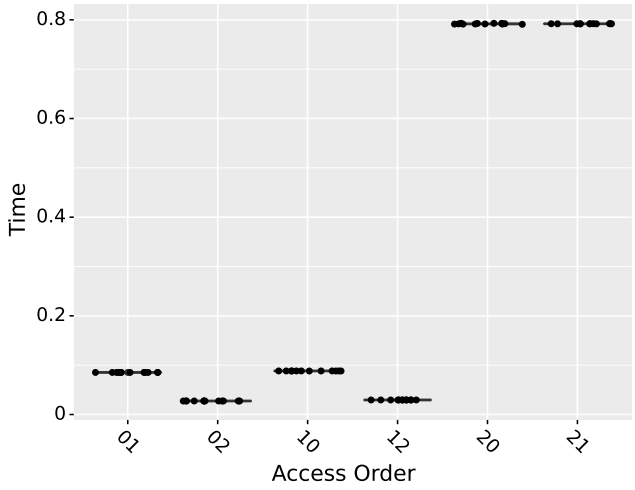
We implemented five variants of each benchmark. The first variant, `RAJA_OpenMP`, is the original RAJA implementation. `RAJALC` modifies `RAJA_OpenMP` to use RAJALC kernel objects. `Hand_Layout`, `Format_Decision`, and `Model_Choice` all augment the `RAJALC` variant to include format changes. Table 2 describes the variants in more detail.

Figure 4 reports the average speedup of the variants relative to the `RAJA_OpenMP` variant. The average is over 100 runs. For the 2MM and 3MM benchmarks, the results show a meaningful speedup for the three variants that make layout changes (7.7% to 8.3% for 2MM and 7.0% to 7.2% for 3MM). For MVT and GEMVER, because no layout changes are made, we do not see meaningful performance improvement (<1% change). Across all benchmarks, there is little difference performance difference among the variants that make layout changes. This indicates that the overhead of our system is minimal and that our model makes comparable layout choices to hand-tuning.

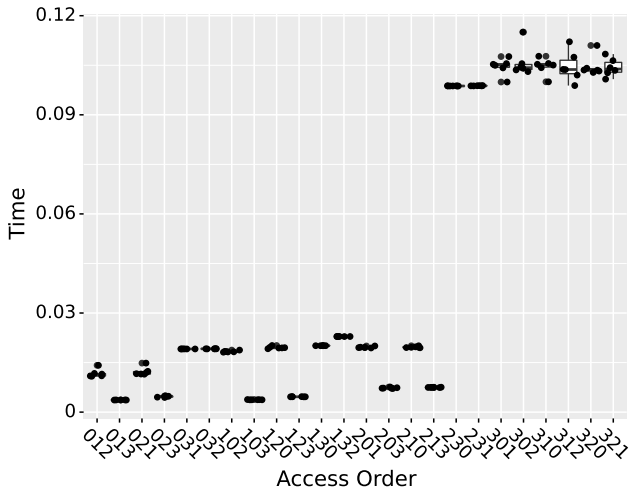
To evaluate the productivity of our contribution, we report the source lines of code added or changed for each of the variants. This value is calculated by counting the number of line differences using python's `Differ`. We preprocess to remove all whitespace-only lines. Figure 5 shows the results. For the variants that make layout changes, we see a large reduction in the amount of changes required using `FormatDecisions` compared to the by-hand implementation. While our tool reduces the code required by more than half, it is important to note that this is with respect to an in-line format conversion. It is possible for a user to themselves write a function that performs the layout change. For the variants without layout changes, we see that using `FormatDecisions` can require more code changes than a by-hand implementation. This is because the `Hand_Layout` variant is identical to the `RAJALC` implementation, while the `Format_Decision` and `Model_Choice` variants instantiate the `FormatDecisions` object.



(a) Box plots showing execution times for 3-dimensional loop accessing 3-dimensional view, grouped by access order. Each point represents a different combination of policy, layout, and argument orders. Each access order group is jittered.



(b) Box plots showing execution times for 3-dimensional loop accessing 2-dimensional view, grouped by access order. Each point represents a different combination of policy, layout, and argument orders. Each access order group is jittered.



(c) Box plots showing execution times for 4-dimensional loop accessing 3-dimensional view, grouped by access order. Each point represents a different combination of policy, and layout for a fixed argument order. Each access order group is jittered.

6.2 Experiment 2: Exhaustive Search for 2mm

Our second experiment further examines the efficacy of our model. Whereas the experiments of in Section 5.2 examined the efficacy of access order representing individual accesses, this experiment considers the model as a whole. For the 2mm benchmark, we evaluated the performance of all possible format choices and compare their execution times against their model scores (the value of the objective function for that choice). We average execution times over five runs. The absolute accuracy of our model is measured by the correlation between the execution time and the model score. However, we are most interested in the *relative* accuracy of our model. A model is more relatively accurate when choices with better model scores have lower execution times, regardless of the specific values of the score or execution time.

Figure 6 plots the model scores and execution times for the different choices. While the overall relationship between model score and execution is as expected, there is one notable outlier, colored in red. This point, which represents the choice that makes no changes to the data's format, has an unusually high model score for its relatively low execution time. This is potentially attributable to our model overestimating the cost of a non-ideal data layout and underestimating the cost of changing formats.

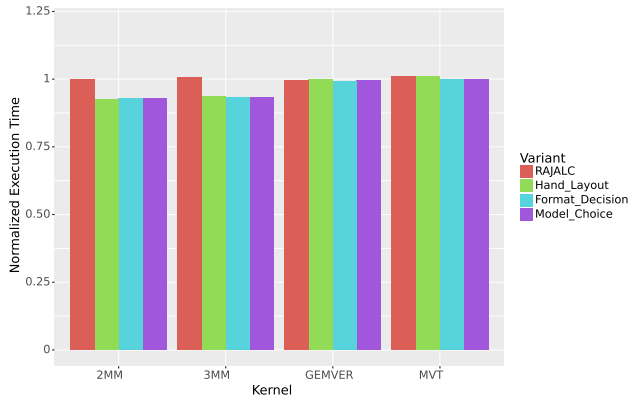


Figure 4: Normalized execution time of the Polybench variants relative to the RAJA_OpenMP variant (Lower is better). Referent execution times are 3.2,4.7,0.039, and 0.12 seconds.

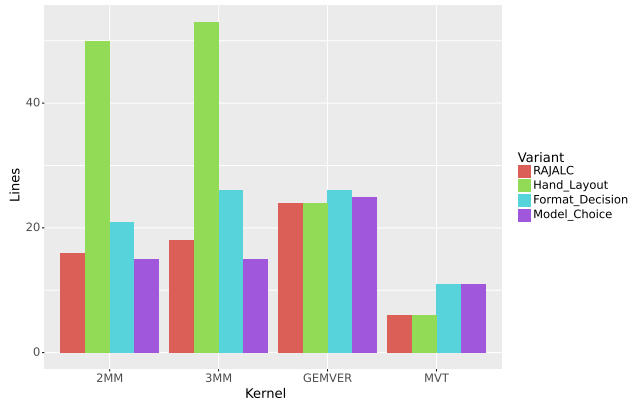


Figure 5: Source lines of code modified (added, changed, or removed) relative to the RAJA_OpenMP variant. (Lower is better)

This experiment also uncovers a limitation of our approach. Notice that many of the different format choices have the same model score. While their execution times do cluster, there is still variability in their distribution. When there are multiple choices with the best model score, the final choice is random amongst them. Future work will need to develop a procedure for making a more intelligent selection among the well-scoring options. This may involve considering the cache interactions of the different references in a kernel.

7 CONCLUSION

We conclude with a number of limitations and directions for future work. The main limitation of the present study is the extent of the evaluation. While four benchmark programs gives a general picture of our technique and its impact, more benchmark kernels need to be examined. Similarly, an evaluation should be completed on a full application rather than only on benchmark loops. Addressing this limitation will require support for more complex loop nests and

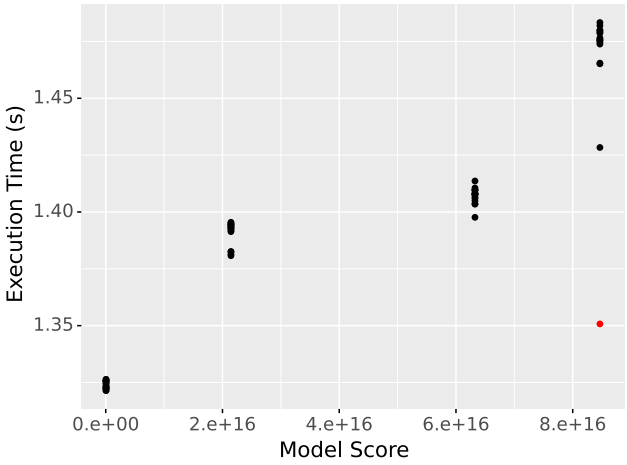


Figure 6: Execution time and model score of all possible format choices for the 2MM kernel. The model selects the choice with the lowest score. Outlier shown in red. All dimension sizes are 1024.

bounds. Another limitation is the fixed nature of our performance model. Future work should incorporate an interface for defining and using different performance models, allowing users to prioritize different aspects of the transformation decision. Currently, the performance model runs with user inputs as strict constraints. While this is a deliberate design decision to give users full control over the optimization, future work should provide warnings if the model identifies a better choice than the one selected by the user. The data layout transformations presented in this paper work only in isolation. An element of future work will support the simultaneous specification of data layout and loop schedule transformations. Lastly, our approach to the data format problem herein only targets different dimension orderings for dense arrays. This technique could be extended to work for sparse arrays as well.

Overall, data format is a key consideration when optimizing applications. Especially in performance portability libraries like RAJA, developers need tools to implement those optimizations without sacrificing productivity. Our API gives developers those tools. By combining declarative user specification of data format with runtime performance modeling, we can offer the performance improvements of data layout changes with significantly less developer effort.

REFERENCES

[1] G. Chen, M. Kandemir, and M. Karakoy. A constraint network based approach to memory layout optimization. In *Design, Automation and Test in Europe*, pages 1156–1161. IEEE, 2005.

[2] G. Chen, O. Ozturk, M. Kandemir, and I. Kolcu. Integrating loop and data optimizations for locality within a constraint network based framework. In *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, pages 279–282. IEEE, 2005.

[3] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. *ACM SIGPLAN Notices*, 30(6):205–217, 1995.

[4] R. E. Diaconescu and H. P. Zima. An approach to data distributions in chapel. *The International Journal of High Performance Computing Applications*, 21(3):313–335, 2007.

[5] H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing*, 74(12):3202–3216, 2014.

[6] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *International Conference on Compiler Construction*, pages 225–245. Springer, 2011.

[7] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, 2001.

[8] R. D. Hornung, H. E. Hones, et al. Raja performance suite. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2017.

[9] R. D. Hornung and J. A. Keasler. The RAJA portability layer: Overview and status. Technical Report Tech. Rep. LLNL-TR-661403, Lawrence Livermore National Laboratory, September 2014.

[10] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 285–296. IEEE, 1998.

[11] K. Kennedy and U. Kremer. Automatic data layout for high performance fortran. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, pages 76–es, 1995.

[12] C. D. Krieger, M. M. Strout, C. Olschanowsky, A. Stone, S. Guzik, X. Gao, C. Bertolli, P. H. Kelly, G. Mudalige, B. V. Straalen, and S. Williams. Loop chaining: A programming abstraction for balancing locality and parallelism. In *Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, May 2013.

[13] S. Kronawitter, S. Kuckuk, H. Köstler, and C. Lengauer. Automatic data layout transformations in the exastencils code generator. *Modern Physics Letters A*, 28(03):1850009, 2018.

[14] M. Kruse and H. Finkel. User-directed loop-transformations in clang. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 49–58. IEEE, 2018.

[15] M. Kruse and H. Finkel. Design and use of loop-transformation pragmas. In *International Workshop on OpenMP*, pages 125–139. Springer, 2019.

[16] F. Luporini, M. Lange, M. Louboutin, N. Kukreja, J. Hückelheim, C. Yount, P. A. Witte, P. H. J. Kelly, G. J. Gorman, and F. J. Herrmann. Architecture and performance of devito, a system for automated stencil computation. *CoRR*, abs/1807.03032, 2018.

[17] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, 1996.

[18] R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, New York, NY, USA, 2015. ACM.

[19] B. Neth, T. R. Scogland, B. R. de Supinski, and M. M. Strout. Inter-loop optimization in raja using loop chains. In *Proceedings of the ACM International Conference on Supercomputing*, pages 1–12, 2021.

[20] O. Ozturk. Data locality and parallelism optimization using a constraint-based approach. *Journal of Parallel and Distributed Computing*, 71(2):280–287, 2011.

[21] L.-N. Pouchet et al. Polybench: The polyhedral benchmark suite. *URL: http://www.cs.ucla.edu/pouchet/software/polybench*, 437, 2012.

[22] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[23] S. Verdoolaege. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*, pages 299–302. Springer, 2010.

[24] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, 1991.