The Design and Implementation of OpenMP 4.5 and OpenACC Backends for the RAJA C++ Performance Portability Layer

William Killian[1,2,3], Tom Scogland[1], Adam Kunen[1], and John Cavazos[3]
{killian4,scogland1,kunen1}@llnl.gov cavazos@udel.edu

[1] Lawrence Livermore National Laboratory, Livermore CA 94550, USA
[2] Millersville University of Pennsylvania, Millersville PA 17551, USA
[3] University of Delaware, Newark DE 19716, USA

Abstract. Portability abstraction layers such as RAJA enable users to quickly change how a loop nest is executed with minimal modifications to high-level source code. Directive-based programming models such as OpenMP and OpenACC provide easy-to-use annotations on for-loops and regions which change the execution pattern of user code. Directive-based language backends for RAJA have previously been limited to few options due to multiplicative clauses creating version explosion. In this work, we introduce an updated implementation of two directive-based backends which helps mitigate the aforementioned version explosion problem by leveraging the C++ type system and template meta-programming concepts. We implement partial OpenMP 4.5 and OpenACC backends for the RAJA portability layer which can apply loop transformations and specify how loops should be executed. We evaluate our approach by analyzing compilation and runtime overhead for both backends using PGI 17.7 and IBM clang (OpenMP 4.5) on a collection of computation kernels.

Keywords: directive-based programming model, performance portability, abstraction layer, code generation

1 Introduction

Directives provide a simple mechanism for annotating source code which provides additional hints to a compiler. OpenMP [12] was one of the first standardized models to leverage directive-based program transformations. Such models enable a user to annotate a region or `for`-loop which allows the compiler to deduce additional information about the program. Compilers with OpenMP support are able to emit parallelized code from source code which had no original notion of being parallelizable.

Due to the increasing demand of heterogeneous computing, OpenACC [11] emerged as a directive-based programming model targeting accelerators including GPUs. OpenACC tends to prefer a more descriptive annotation model where a user may not need to explicitly state *how* a loop should be executed. Instead,

the user indicates to the compiler that a loop *can* be parallelized. On the other hand, OpenMP leans toward a more prescriptive annotation model where a user must clearly state how a loop should be executed. The OpenMP standards committee released OpenMP 4.0 and 4.5 which enabled and improved upon [13] heterogeneous targets. Both programming models are interesting in the scope of this research due to (1) their directive-based approach of parallelization, (2) cross architecture support, and (3) targeting heterogeneous systems.

Performance Portability Layers make a limited set of assumptions about programs and allow a user to represent a program as an embedded domain specific language. Although this language has reduced usage compared to a general-purpose programming model, the portability layer is able to make additional assumptions about a user's code and apply high- and low-level source code transformations. This is incredibly useful when a user may wish to explore an unknown optimizations search space or compare/contrast multiple programming models.

There are many portability layers in active and maintained development. Thrust [1] is a C++ library which enables users to easily target GPUs and multi-core CPUs without needing to know CUDA [10], NVIDIA's proprietary programming language for targeting GPUs. Agency [9] is a relatively new portability layer which leverages C++ templates to drive parallel programming. The primary difference between Agency and Thrust lies with the level of abstraction – Agency provides much more fine-grained control over *how* and *where* to execute.

Kokkos [2] is another C++ library with an interest in unifying data parallelism and memory access patterns. Kokkos is capable of restructuring data at compile time with their generic *View* concept which helps improve performance across different target architectures, such as CPUs, GPUs, and many-core architectures. Kokkos also provides many different execution policies to end users, giving them enough flexibility to experiment with competing programming models (e.g. OpenMP 4.5 and CUDA).

More recently, the C++ Standards Committee (WG21) approved parallel Standard Template Library functions as part of the Parallelism TS [5]. They augmented many of the algorithms found under `algorithm` and `numeric` to include *execution policy* types. The standard defines three types: `seq` (sequential, ordered), `par_seq` (parallelized but in sequenced order), and `par_useq` (parallelized, unsequenced ordering) [6]. Vendors are also able to provide their own execution policies. Intel [4], SYCL [8], and HPX [7] were the first entities to release feature-complete versions of the Parallelism TS publicly.

The Khronos group has also come out with SYCL, a C++ single-source heterogeneous programming model for OpenCL [8]. SYCL provides a cross-platform API which enables code to be written in C++ and target any OpenCL device.

RAJA [3] is another C++ library which provides many different types of *execution policies*. One advantage to RAJA over other portability layers is the support for a variety of backends, ranging from sequential, SIMD, and OpenMP on CPU architectures to CUDA, OpenMP 4.5, and (with this research) OpenACC on GPU architectures. In this research we extend our OpenMP backend and

propose an initial OpenACC backend implementation. We outline the concepts and key structures of RAJA in Section 2. Then we introduce our type framework for creating execution policies in Section 3. We then outline the specializations required for OpenMP 4.5 and OpenACC in Sections 4 and 5. Finally, we evaluate the overhead in terms of (1) compilation time and (2) execution time. We leverage a collection of kernels, both hand written and automatically generated RAJA versions, to evaluate the overhead in using a portability abstraction layer outlined in Section 6.

2 RAJA

RAJA can be reduced to three relevant concepts for the scope of this research. These three concepts define the expected interfaces, types, and pre- and post-conditions when library writers and users leverage RAJA to generate target code from a portable interface.

`Execution Policy` First, an execution policy in RAJA tells the compiler which code path to follow and expand when expanding on loop and loop nests. RAJA provides many different backend targets, including sequential, OpenMP, SIMD, CUDA, and OpenACC implementations. Depending on the type of execution policy, the code which is conditionally enabled may change. Execution policies ultimately drive the required code transformations and generation necessary to provide a user with the expected behavior of specifying a given policy.

`Callable` Second, a callable is a lambda function or a function which is invocable with a specified number of arguments. The *Callable* is also known as the loop body of a loop nest within the scope of RAJA. The callable must not be *mutable*. Depending on the target backend, the *Callable* may need additional attributes specified, such as `__host__ __device__` attribute indicators for CUDA.

`RandomAccessContainer` Third, a random-access container defines an *iteration space* for a given range of elements. The functional requirements of a RandomAccessContainer are:

- must have `begin()` and `end()` which each return *RandomAccessIterator*s to the underlying type
- must have `size()` to indicate the size of the container
- the underlying type (`decltype(*container.begin())`) must be convertible to a specified *Callable* lambda function.

2.1 Basic Execution Policies

The most basic executor in RAJA is a simple for-loop executor in the current thread. No code transformations or directives are emitted in this base case. Instead, RAJA will just create a for-loop which iterates over the *RandomAccessContainer* specified in the `forall` call and invoke the *Callable* function with the

underlying value of each element contained within the *RandomAccessContainer*. More complex executors may augment or change the code path to include the emission of directives or other code. Section 3.3 outlines the specialization of `forall` for arbitrary execution policies.

2.2 `RAJA::NestedPolicy` and loop transformations

RAJA provides a powerful loop transformation construct, `forallN`. `forallN` is similar to `forall`, but enables the user to specify loop transformations for nested loops such as permutation and tiling. Any combinations of permutations and tiles can be applied. The `NestedPolicy` type is composed with an `ExecList` followed by any number of `Tile` clauses. `ExecList` describes the execution policy to apply to each loop nest. `Tile` clauses can specify a `TileList` to apply tiles of specified sizes to each loop. Permutations can also be specified with a `Permute` clause. Multiple `Tile` clauses can be applied to a single loop nest, making it feasible to optimize for outer- and inner-cache data access patterns. An example `NestedPolicy` is shown in Listing 1.

```
using Policy = NestedPolicy<
  ExecList<
    omp_collapse_nowait_exec,
    simd_exec,
    omp_collapse_nowait_exec>,
  OMP_Parallel<
      Tile<TileList<
        tile_none,
        tile_fixed<8>,
        tile_fixed<32>>,
      Permute<idx_seq<1,2,0>>>>>;
```

Listing 1: Sample `NestedPolicy` for use within `RAJA::forallN`. Note that multiple tiling policies may be specified in addition to outer "wrap" policies which can create regions of code. Loops may also be permuted to an arbitrary ordering, and, provided backend support, loops may also be collapsed

RAJA provides specializations for `omp_for_nowait_exec` policies that are next to one another. After any permutation of loops is applied, RAJA will look for any number of adjacent `omp_collapse_nowait_exec` policies. If two or more `collapse` policies are directly adjacent, a single *`#pragma omp for collapse(N)`* directive is emitted above the N next loops.

3 Embedding Directives in the C++ Type System

In general, most execution-based directives are applied to `for`-loops. RAJA provides the `forall` and `forallN` loop constructs to specify an execution policy on

a single loop or loop nest. The most challenging component of this research is embedding the required directive information in a C++ type to apply *template specialization* with careful consideration of SFINAE[4]. To tackle this embedding problem, we make the following translations:

- All supported directives have a functional mapping to a single type; not all clauses and options are supported.
- OpenACC/OpenMP `parallel` and OpenMP 4.5 `target` enclose *regions*, or structured code blocks
- OpenACC `loop`, OpenMP 4.5 `teams distribute`, and OpenMP `for` constructs are *variadic* – zero or more clauses may be added to one of these constructs.

The construction of a type system for a directive-based programming language can be described with the following steps:

1. Define all valid constructs, directives, and clauses as *policy tags*
2. Construct explicit types for each construct, directive, and clause
   - When defining a *construct* that encloses a region, an `inner` type must be defined. The appropriate region code shall be emitted and the inner policy will then be invoked.
   - When defining a *construct* or *directive* which may have clauses optionally specified, the new type must accept a variadic number of template argument types and inherit from all specified variadic template arguments.
   - Clauses with value-based options, e.g. `num_gangs`, must be defined with a uniquely-identifiable `static constexpr int` member variable.
3. Implement all specializations for the backend planned for support

Below, we highlight a trivial case of supporting OpenMP 3.x `parallel for` with an optionally specified `schedule` clause.

3.1 Defining Policy Tags for a Backend

Tags are defined within a nested `tags` namespace for the supported backend. For the case of an `OpenMP` policy, we will make use of the `omp` namespace. Listing 2 shows the definitions of various tags used to build OpenMP policies. There are two primary types of tags. The first – region-based tags – describe tags which are used to define a region containing another policy. The second type of tag, construct- and clause-based tags, describe all other valid policy tags present for a given backend (e.g. OpenMP). The aggregation of these tags within a single C++ type is how we specialize `forall` for a backend implementation.

[4] Substitution Failure Is Not An Error – the C++ standard states that substitution failure shall not result in a compiler error unless no valid substitutions are found

```
namespace omp::tags {
  // region-based tags
  struct Parallel {};
  struct BarrierAfter {};
  // ... BarrierBefore, etc.
  // construct- and clause-based tags
  struct For {};
  struct Static {};
  // ... Guided, Dynamic
}
```

Listing 2: Defining tag types for the OpenMP 3.x `parallel for` clause

3.2 Constructing Explicit Execution Policy Types

Once the policy tags are defined, we construct our explicit types for directives and clauses. Each of these explicit types defines a component of a *policy*. A *policy* defines how the code should be analyzed by a generator or specialization. Within the scope of RAJA, a policy must define (1) a type (e.g. *sequential*, *OpenMP*) and (2) a platform (e.g. *undefined*, *cpu*, *gpu*). The type is useful for determining whether a constructed policy is valid. The platform information is used by the optional data integration layer to automatically perform data transfer. In Listing 3 we show the definitions of a `PolicyBase` type used to construct subsequent execution policies.

```
enum class Policy { seq, simd, cuda, openmp, target_openmp, openacc };
enum class Platform { undefined, cpu, gpu };

template <Policy Pol, Platform P, typename... Args>
struct PolicyBaseT : public Args... {
  static constexpr Policy policy     = Pol;
  static constexpr Platform platform = P;
};

// specific policy alias type for all OpenMP policies
namespace omp {
template <typename... Args>
using policy = PolicyBaseT<Policy::openmp, Platform::undefined, Args...>;
}
```

Listing 3: Base policy types and platforms used to construct additional policies in RAJA. Note that the specialized OpenMP type alias has no defined platform to permit reuse between OpenMP 3.x and OpenMP 4.5 target offload execution policies.

Region-like directives (e.g. `omp parallel` in OpenMP 3.x) define a code block to enclose. A generic region policy must represent a code region within the type system. We accomplish this by defining an *inner policy* type. This *inner policy* type indicates how the code within the region should execute with another execution policy. Listing 4 shows the construction of a Parallel policy within the `omp` namespace. Once the parallel execution policy is defined, it can be used in conjunction with other policies to drive source code transformations at compile time.

```
namespace omp {
template <typename Inner>
struct Parallel : policy<tags::Parallel> {
  using inner = Inner;
};
}
```

Listing 4: Construction of an OpenMP 3.x parallel region policy

Other directives, such as `omp for` define how an immediately-following `for`-loop should be distributed within the current parallel region. Additional clauses can change the execution behavior of the `for`-loop, specifically `schedule` and `nowait`. Listing 5 shows how a `for`-loop directive is constructed. Specifying additional options for an OpenMP for execution policy is not required, but a policy clause still must indicate any required information through the type system. The specialization of the `schedule(static,N)` clause is shown in Figure 6. In addition to a template argument specifying the static chunk size, a `static constexpr int` member variable with a uniquely identifiable name is defined as an accessor for the template argument value. Additional clauses, including guided and dynamic scheduling clauses, are omitted for brevity.

```
template <typename... Options>
struct For : policy<tags::For>, Options... {};
```

Listing 5: Construction of an OpenMP for policy with template arguments. Note that the `Options` are variadically inherited to encode the underlying policy a "parent" to the current policy.

3.3 Implement `forall` Specializations

Once all directives and clauses are defined, we can implement `RAJA::forall` specializations with *aggregate* policies. An aggregate policy is a fully-defined policy which should be made available to end users. RAJA provides a few aggregate

```
template <unsigned int N>
struct Static : policy<tags::Static>  {
  static constexpr unsigned int static_chunk_size = N;
};
```

Listing 6: Definition of an OpenMP static schedule clause for an OpenMP for directive. Since the static schedule has no other options besides the chunk size, there is no additional inheritance besides the policy definition.

```
using omp_for_exec = omp::BarrierAfter<omp::For<>>;

template <unsigned int N>
using omp_static_exec = omp::BarrierAfter<omp::For<omp::Static<N>>>;

using omp_for_nowait_exec = omp::For<>;

template <unsigned int N>
using omp_static_nowait_exec = omp::For<omp::Static<N>>;

template <typename Inner>
using omp_parallel_exec = omp::Parallel<Inner>;

using omp_parallel_for_exec = omp_parallel_exec<omp_for_exec>;

template <unsigned int N>
using omp_parallel_static_exec = omp_parallel_exec<omp_static_exec<N>>;
```

Listing 7: Aggregate policy definitions for the RAJA OpenMP 3.x backend. Note that the default `for` execution policy is `nowait` and all aggregate policies are type aliases; no new types are introduced.

policies for OpenMP 3.x, e.g. `omp_parallel_for`, `omp_parallel_for_nowait`, `OMP_Parallel`. An aggregate policy can be viewed as a type alias to a nesting of policy directives and clauses. Figure 7 shows a subset of aggregate policies implemented for the RAJA OpenMP 3.x backend.

We implement corresponding `RAJA::forall` specializations for each defined aggregate policy. Given the type hierarchy of an aggregate policy, we can determine all enclosed directives and clauses in the following manner:

1. α is the set of all possible directives/clauses valid for the current backend.
2. ϵ is the set of directives/clauses required for a `forall` specialization.
3. β is the set of directives/clauses present in a given execution policy.
4. Ensure $\beta \cup \epsilon = \beta$.
5. Ensure $\beta \cup (\alpha \setminus \epsilon) = \emptyset$.
6. Use SFINAE-safe conditional visibility of the `forall` specialization by restricting visibility of the specialization, ϵ, to policies equivalent to β.

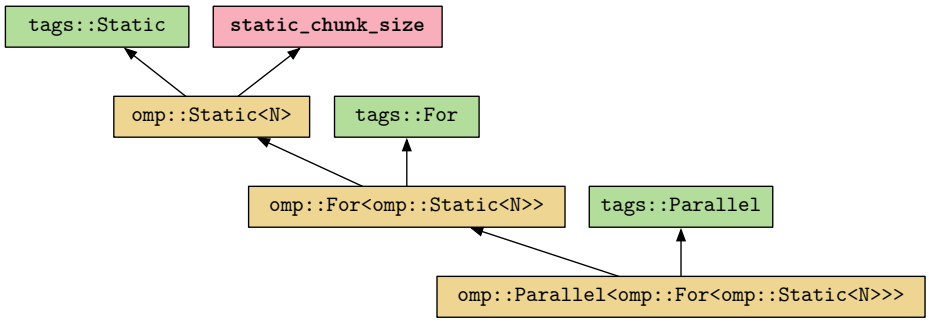

Fig. 1. The full type hierarchy of an instantiated `omp parallel for schedule(static, N)` policy. Red indicates a `static constexpr int` member variable, green indicates a *tag* type, and yellow represents a *policy* type.

Fig. 2. The flattened type hierarchy of an instantiated `omp parallel for schedule(static, N)` policy. The specialization is easy to deduce from the specified tags (shown in green). Any attributes needed by the specialization can be accessed with *field names* (shown in red).

We show an example type hierarchy of an OpenMP Parallel for directive with a fixed, static schedule in Figure 1. Furthermore, when we collapse the type hierarchy into a flat view, depicted in Figure 2, we can easily check the constraints to determine the most specialized valid execution.

This constraint satisfaction can easily be implemented in C++ by counting the occurrences of β in both α and ϵ and ensuring they are the same. The `exact<T...>` metafunction provides an alias to `std::enable_if<T...>::type` if and only if the constraint is met. An additional layer of dispatch to the generic `forall` interface exists to generalize to a given backend. Listing 8 shows a subset of `forall` specialization for the OpenMP 3.x backend as well as the `forall` dispatch to an OpenMP policy type.

4 Case Study: OpenMP 4.5

In addition to supporting `parallel for` and other clauses from the OpenMP 3.x standard, we extend RAJA to support a number of OpenMP 4.5 directives and clauses. From a design perspective, we chose to extend the OpenMP implementation as follows:

1. Extend the tags to include `Target`, `Teams`, and `Distribute`
2. Define aggregate policies for `TargetTeamsDistribute`, `TargetTeamsDistributeParallelFor`, and `TargetTeamsDistributeParallelStatic`
3. Define a dispatch overload for `forall` with all OpenMP 4.5 policies

4. Define built-in policies for some OpenMP 4.5 policies: `OMP_Target` (for `forallN`), `omp_target_teams_distribute_parallel_for`, and a few others.

Listing 9 highlights the augmentation of OpenMP 3.x tags to OpenMP 4.5. We first include all of the tags under the `omp::tags` namespace, then we introduce the additional tags required by OpenMP 4.5. Once all of the tags are defined, we need to construct the *tag list*. The *tag list* is used to determine the most specialized and valid version of an execution policy.

Once all tags are defined, we can create the aggregate policies to allow for proper OpenMP 4.5 team distribution on GPUs. Listing 10 shows the creation of the aggregate policies. Each aggregate policy requires the number of teams to be specified at a template parameter. Additionally, there is one aggregate policy which also expects a static chunk size to be specified as an additional template parameter. By configuring the sizes as template parameters, we can ensure (1) the specialization can be encoded into a C++ custom data type and (2) the compiler is aware of the sizes at compile time.

Finally, once all aggregate policies are created, we can implement the `forall` overloads within the OpenMP 4.5 backend. Listing 11 shows the overload for one of the aggregate policies and the dispatch code expecting a policy and potentially invoking the OpenMP 4.5 policy.

5 Case Study: OpenACC

OpenACC provides a different set of directives, constructs, and clauses for RAJA to consider with code generation. First, OpenACC allows two different types of region constructs: `parallel` and `kernels`. `parallel` is more synonymous with the OpenMP 4.5 `parallel` construct while `kernels` provides a much higher-level annotation of a loop nest. The compiler is ultimately able to make many more decisions regarding code optimization and transformation when given a `kernels` construct compared with a `parallel` construct. The increased number of clauses present in OpenACC also mandate additional tag definitions as depicted in Listing 12. Like other backends, some tags are only valid in certain contexts. Therefore, a program is illformed if a user specifies an invalid policy construction.

Listing 13 shows definitions for a subset of OpenACC aggregate policies. `Parallel` and `Kernels` are scope-based policies which do not generate any loop iteration code. Policies such as `NumGangs` and `VectorLength` require template parameters indicating their size but will always be additional clauses specified for the scope-based policies. Likewise, `Independent`, `Gang`, and `Worker` policies have no template arguments but must be specified as additional clauses for a `Loop` construct.

Ultimately, the grammar defined by the OpenACC standard is adhered to through the established constraints of our type system. Only valid execution policies have specializations implemented. Listing 14 shows a subset of specializations defined in our OpenACC backend. It is important to note that the

number of specializations we need is a function of the configuration parameters which can either exist or not exist for a given construct. Since `Kernels` can have any number of three clauses specified (`NumGangs`, `NumWorkers`, `VectorLength`), we must implement 8 versions of kernels. Likewise, the `Loop` construct can have any number of four clauses specified (`Independent`, `Gang`, `Worker`, `Vector`), resulting in 16 versions required.

6 Evaluation

Our evaluation system is a CORAL Early Access System which has two IBM POWER 8 processors and four NVIDIA P100 GPUs with NVLINK. The processor, a 10-core IBM POWER 8+, has a core frequency of 4.0GHz. Two processors are on each node and is coupled with 256GB of DDR3 RAM. Only one NVIDIA P100 (SMX variant) was used for consistent experimentation. For all of our tests we restrict execution to GPU device 0 and leverage unified memory for data allocation and offload. When using the proposed OpenACC backend, we compile our test set with PGI Compiler 17.7[5]. With the OpenMP 4.5 backend, we compile our test set with IBM's version of clang with OpenMP 4.5 support. Both of these compilers and supporting libraries leverage CUDA 8.0.61.

6.1 Test Set

We used a collection of various kernels which highlight different access patterns (dense linear algebra, stencils, and reductions of k-dimensions to $k-1$ dimensions). One limitation of our evaluation is the absence of any reductions. To this end, RAJA cannot provide support for directive-based reductions because the variable name is required within a directive reduction clause. Because of this limitation, inclusion of directive-based reductions is out of scope for this research – it is impossible to generate the directives with library-only solutions (e.g. RAJA). RAJA does have support for reductions, but it is achieved using allocated arrays and specialized kernels and combiners. OpenMP 4.5 reducers are implemented and are currently being used, but we wanted to focus on execution policies instead of reducers.

[5] This work was not feasible until the release of V17.7 in early August which added support for lambdas and no-copy captures

Kernel	Description	OpenACC	OpenMP
Jacobi1D	1-D Stencil	64	16
Jacobi2D	2-D Stencil	256	64
Heat3D	3-D Heat Equation	1024	256
MatMul	Matrix Multiplication	1024	256
MatVec	Matrix-Vector Multiplication	256	64
VecAdd	Vector Addition	64	16
Tensor2	Synthetic tensor contraction (2D to 1D)	256	64
Tensor3	Synthetic tensor contraction (3D to 2D)	1024	256
Tensor4	Synthetic tensor contraction (4D to 3D)	4096	1024

Shown above is the test set we end up comparing compiler overhead of RAJA-based versions to directive-based implemented versions. There is a larger "tuning" space for OpenACC versions due to the expanded clause options for `loop` directives compared to the clause options available for `parallel for` and `teams distribute` directives

6.2 Goals and Non-Goals

It is our goal in this evaluation section to highlight:

- Compare the compile-time overhead of leveraging meta-programming concepts and C++ templates to drive code generation to directive-based approaches
- Compare the code generation of RAJA-fied loop nests to hand-written directive-based loop nests

In no way to we intend to highlight the following:

- Compare the two compilers' performance on the same kernel directly
- Compare OpenACC to OpenMP 4.5 in terms of:
  - Compilation time
  - Compilation resources (RAM, page faults, etc)
  - Code generation
  - Execution time
- Identify bottlenecks of toolchains without vendors' knowledge of the problem
- Determine any limitations of drivers, software, hardware, or runtimes.

6.3 Compilation Overhead

For each kernel we show the compilation overhead. We used a wall clock timer to measure the total amount of time it took to compile each set of kernels with a given compiler. In the following table we highlight (1) the kernel name, (2) the average compilation overhead when compiling the various versions of the kernel, and (3) the backend used. The overhead is computed by taking the difference

in compilation times over the directive-based compilation time. The average overhead with the OpenACC backend is 95.07% while the average overhead with the OpenMP backend is 38.78%.

Kernel	**OpenACC**	**OpenMP**
Jacobi1D	17.50%	8.75%
Jacobi2D	50.24%	20.42%
Heat3D	74.40%	30.91%
MatMul	80.28%	31.24%
MatVec	45.41%	16.47%
VecAdd	15.20%	6.24%
Tensor2	48.94%	17.57%
Tensor3	72.85%	27.53%
Tensor4	120.74%	59.29%
Average	*95.07%*	*38.78%*

Compilation overhead of test programs using a directive-based RAJA backend instead of manually-specified directives

There are somewhat significant changes between our original kernels and RAJA kernels. First, The original kernels are essentially C functions. The first change that the RAJA versions make is converting from traditional for-loops to RAJA *Iterables* and lambda expressions. A simple two-nested for-loop would go from zero template instantiations to at least three (two for each *Iterable* and one for the execution policy). When the code passes through the compiler for code generation and specialization, the execution policy will dictate overload visibility from substitution failure. In the case of a RAJA *forall* construct, we will attempt to specialize for each possible backend (sequential, SIMD, OpenMP, OpenACC, OpenMP, TBB, CUDA, etc). Determining the code path would require traversing through all possible backends results in over 8 resolution attempts. Once a policy is determined, then the per-backend specializations are evaluated and the correct version is visited for code emission. The *Iterable* must also be validated which adds two additional overload resolution checks per nest level. The current RAJA *forallN* nested loop construct will construct at least 5 additional types per nest level. On a three-nested loop, at least 18 type constructions and 110 overload resolution attempts will be made with an OpenACC execution policy. For OpenMP 4.5 the overload resolution number reduces down to 62. Comparing this to the zero type constructions and no overload resolutions from the original kernel versions provides some additional insight as to where the compilation overhead comes from.

6.4 Runtime Overhead

Next, we show the runtime overhead for each kernel. We used a timer to measure the total amount of time it took to execute the kernel *on the GPU* with each

set of kernels and a given compiler. In the following table we highlight (1) the kernel name, (2) the average execution overhead when executing the various versions of the kernel, and (3) the backend used. The average overhead with the OpenACC backend is 1.66% while the average overhead with the OpenMP backend is 1.69%. This overhead differs greatly from the compilation overhead, suggesting that although compilation takes significantly longer, the emitted code performs about the same as the plain directives. One of the underlying goals of performance portability layers, such as RAJA, is to minimize the execution overhead.

Kernel	**OpenACC**	**OpenMP**
Jacobi1D	2.52%	1.94%
Jacobi2D	1.25%	1.14%
Heat3D	1.08%	1.19%
MatMul	0.96%	1.01%
MatVec	1.13%	1.38%
VecAdd	0.21%	0.38%
Tensor2	0.98%	1.21%
Tensor3	1.34%	1.44%
Tensor4	2.18%	2.14%
Average	*1.66%*	*1.69%*

Runtime overhead of test programs using a directive-based RAJA backend instead of manually-specified directives

7 Future Work and Conclusion

In this research we propose a backend design and implementation which provides a subset of OpenMP 4.5 and OpenACC to users of the RAJA portability layer. We address concerns related to template specialization and overloading, version explosion, compilation overhead, and runtime overhead. We highlight the various components of our implementation including execution policy dispatch, specialization for regions, and aggregation for various clause combinations found within OpenMP and OpenACC.

We show that with the OpenMP 4.5 backend compiled with the IBM clang we observe – on average – a 40% slowdown in compilation time but only a 1.69% slowdown in execution time compared to directive-only based implementations of the test programs. When using the OpenACC backend compiled with PGI 17.7 we observe – on average – a 95% slowdown in compilation time but only a 1.66% slowdown in execution time compared to directive-only based implementation of the test programs. We attribute most of the compilation slowdown to the compiler needing to (1) instantiate many more templates compared to the directive-based solutions and (2) perform template overload resolution to find the most specific and valid version of policies. We plan to continue this research by:

- Augmenting our current reduction implementation for OpenMP 4.5 and extending it to the OpenACC backend
- Add more directives and clauses to the backends, specifically some of the `if()` constructs being added with OpenMP 5.0. This would help reduce the total number of specializations required
- Reduce the number of template instantiations necessary by simplifying the check for the most specific policy
- Expand our results to include reductions and other high-level source code transformations (collapsing for OpenACC and OpenMP 4.5, tiling for OpenACC)

References

1. Bell, Nathan and Hoberock, Jared. *Thrust: A Productivity-Oriented Library for CUDA*. GPU Computing Gems. 2011.
2. Edwards, H. Carter and Trott, Christian R. and Sunderland, Daniel. *Kokkos: Enabling manycore performance portability through polymorphic memory access patterns*. Journal of Parallel and Distributed Computing. 2014.
3. Hornung, R. D. and J. A. Keasler. *The RAJA Portability Layer: Overview and Status*. No. LLNL-TR-661403. Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2014.
4. Intel Corporation. *Getting Started with Parallel STL*. March 2017. `https://software.intel.com/en-us/get-started-with-pstl`
5. ISO/IEC. *Programming Languages – Technical Specification for C++ Extensions for Parallelism*. May 2015. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4507.pdf`
6. ISO/IEC. *"Working Draft, Standard for Programming Language C++"*. July 2017. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4687.pdf`
7. Kaiser, Hartmut, et al. *"HPX: A task based programming model in a global address space."* Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models. ACM, 2014.
8. Khronos OpenCL Working Group. *SYCL Provisional Specification Version 2.2*. February 2016. `https://www.khronos.org/registry/SYCL/specs/sycl-2.2.pdf`
9. NVIDIA Corporation. *Agency 0.1.0*. 2016. `https://agency-library.github.io/0.1.0/index.html`
10. NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. June 2017. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`
11. OpenACC Standard Committee. *The OpenACC Application Programming Interface Version 2.5*. October 2015. `https://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5.pdf`
12. OpenMP Architecture Review Board. *"OpenMP Application Program Interface Version 3.0"*. May 2008. `http://www.openmp.org/mp-documents/spec30.pdf`
13. OpenMP Architecture Review Board. *"OpenMP Application Program Interface Version 4.5"*. Nov 2015. `http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`
14. Stone, John E. and Gohara, David and Shi, Guochun. *OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems* Computing in Science & Engineering Vol: 12, Issue: 3. IEEE, 2010.

```
namespace omp {
// all tags for the given backend shall be listed in tag_list
using tag_list = list<tags::Parallel, tags::Static,
  tags::BarrierAfter, tags::For>;

template <typename Exec, typename Iterable,
  typename Body, typename TagList>
exact<Exec, TagList, tags::For, tags::Static>
forall_impl(const Exec &&, Iterable && iter, Body && body) {
  auto size = iter.size();
  #pragma omp for schedule(static, Exec::static_chunk_size)
  for (decltype(size) i = 0; i < size; ++i)
    body(*(iter.begin() + i));
}

template <typename Exec, typename Iterable,
  typename Body, typename TagList>
exact<Exec, TagList, tags::Parallel>
forall_impl(const Exec &&, Iterable && iter, Body && body) {
  #pragma omp parallel
  {
    forall(typename Exec::inner(),
      std::forward<Iterable>(iter), std::forward<Body>(body));
  }
}
}

template <typename Exec, typename Iterable,
  typename Body, typename TagList>
exact<Exec, TagList, tags::BarrierAfter>
forall_impl(const Exec &&, Iterable && iter, Body && body) {
  forall(typename Exec::inner(),
    std::forward<Iterable>(iter), std::forward<Body>(body));
  #pragma omp barrier
}
}

// dispatch to target omp policy
template <typename Exec, typename Iterable, typename Body>
typename std::enable_if<Exec::policy == Policy::openmp>::type
forall(const Exec &&p, Iterable && iter, Body && body) {
  omp::forall_impl<Exec, omp::tag_list>(
    std::forward<const Exec>(p),
    std::forward<Iterable>(iter),
    std::forward<Body>(body));
}
```

Listing 8: `forall` implementations for a subset of OpenMP 3.x specializations.

```
namespace target_omp {
namespace tags {
// include all tags from namespace omp::tags
using namespace omp::tags;
struct Target {};
struct Teams {};
struct Distribute {};
} // end namespace tags

using tag_list =
  list<tags::Target, tags::Teams, tags::Distribute, tags::Parallel,
       tags::Static, tags::BarrierAfter, tags::BarrierBefore, tags::For>;
} // end namespace target_omp
```

Listing 9: Tag definitions for OpenMP 4.5. Reuse of tags from OpenMP 3.x reduces the overall implementation size with minimal cost.

```
namespace target_omp {
template <typename... Args>
using policy = PolicyBaseT<Policy::target_openmp, Platform::undefined, Args...>;

using omp::BarrierAfter;
using omp::BarrierBefore;
using omp::For;
using omp::Parallel;
using omp::Static;

template <unsigned int N>
struct TargetTeamsDistribute : policy<tags::Target, tags::Teams, tags::Distribute> {
  constexpr static unsigned int num_teams = N;
};

template <unsigned int N>
struct TargetTeamsDistributeParallelFor
    : policy<tags::Target, tags::Teams, tags::Distribute, tags::Parallel, tags::For> {
  constexpr static unsigned int num_teams = N;
};

template <unsigned int N, unsigned int M>
struct TargetTeamsDistributeParallelStatic
    : policy<tags::Target, tags::Teams, tags::Distribute, tags::Parallel,
             tags::For, tags::Static> {
  constexpr static unsigned int num_teams  = N;
  constexpr static unsigned int static_chunk_size = M;
};
} // end namespace target_omp
```

Listing 10: Aggregate policy definitions for OpenMP 4.5. In addition to all OpenMP 3.x policies, three new policies are added to aid with target offload

```
namespace target_omp {
template <typename Exec, typename Iterable,
  typename Body, typename TagList>
exact<Exec, TagList, tags::Target, tags::Teams,
  tags::Distribute, tags::Parallel, tags::For>
forall_impl(const Exec &&, Iterable && iter, Body && body) {
  auto size = iter.size();
  #pragma omp target teams distribute parallel for \
    num_teams(Exec::num_teams)
  for (decltype(size) i = 0; i < size; ++i) {
    body(*(iter.begin() + i));
  }
}
} // end namespace target_omp

template <typename Exec, typename Iterable, typename Body>
typename std::enable_if<Exec::policy == Policy::target_openmp>::type
forall(const Exec &&p, Iterable && iter, Body && body) {
  target_openmp::forall_impl<Exec, omp::tag_list>(
    std::forward<const Exec>(p),
    std::forward<Iterable>(iter),
    std::forward<Body>(body));
}
```

Listing 11: A `forall` specialization shown for OpenMP 4.5. Note that the second function performs tagged dispatch of a policy to an OpenMP 4.5 policy if the type matches.

```
namespace openacc {
namespace tags {
struct Parallel {};
struct Kernels {};
struct Loop {};
struct Independent {};
struct Gang {};
struct Worker {};
struct Vector {};
struct NumGangs {};
struct NumWorkers {};
struct VectorLength {};
} // end namespace tags

using tag_list =
  list<tags::Parallel, tags::Kernels, tags::Loop, tags::Independent,
       tags::Gang, tags::Worker, tags::Vector, tags::NumGangs,
       tags::NumWorkers, tags::VectorLength>;
} // end namespace openacc
```

Listing 12: Tag definitions for OpenACC. Note the difference between `Gang` and `NumGangs` – the former indicates a clause on a `loop` construct while the latter specifies the number of gangs on a `parallel` or `kernels` construct.

```
namespace openacc {
template <typename... Args>
using policy = PolicyBaseT<Policy::openacc, Platform::gpu, Args...>;

template <unsigned int N>
struct NumGangs : policy<tags::NumGangs> {
  static constexpr unsigned int num_gangs = N;
};

template <typename Inner>
struct Parallel : policy<tags::Parallel>, Inner {
  using inner = Inner;
};

struct Independent : policy<tags::Independent> {};

template <typename ... Options>
struct Loop : policy<tags::Loop>, Options... {};

} // end namespace openacc
```

Listing 13: A subset of aggregate policy definitions for OpenACC. The user can directly construct their own RAJA OpenACC type policy or leverage one of the policies RAJA provides.

```
namespace openacc {
template <typename Exec, typename Iterable,
  typename Body, typename TagList>
exact<Exec, TagList, tags::Parallel, tags::NumGangs>
forall_impl(const Exec &&, Iterable && iter, Body && body) {
  #pragma acc parallel num_gangs(Exec::num_gangs)
    forall(Exec::inner(),
      std::forward<Iterable>(iter),
      std::forward<Body>(body));
}

template <typename Exec, typename Iterable,
  typename Body, typename TagList>
exact<Exec, TagList, tags::Kernels, tags::NumGangs, tags::VectorLength>
forall_impl(const Exec &&, Iterable && iter, Body && body) {
  #pragma acc kernels num_gangs(Exec::num_gangs) \
      vector_length(Exec::vector_length)
    forall(Exec::inner(),
      std::forward<Iterable>(iter),
      std::forward<Body>(body));
}

// A total of 8 specializations are required for each of tags::Parallel
// and tags::Kernel. Only one is shown for each above.

template <typename Exec, typename Iterable,
  typename Body, typename TagList>
exact<Exec, TagList, tags::Loop, tags::Independent, tags::Vector>
forall_impl(const Exec &&, Iterable && iter, Body && body) {
  auto size = iter.size();
  #pragma acc loop independent vector
  for (decltype(size) i = 0; i < size; ++i) {
    body(*(iter.begin() + i));
  }
}

// A total of 16 specializations are required for tags::Loop
// Only one is shown above.

} // end namespace openacc
```

Listing 14: `forall` specializations for OpenACC. Note we must provide specializations for any number of `kernels` or `parallel` constructs with the {`num_gangs`, `num_workers`, `vector_length`} clauses either (1) being specified or (2) omitted. The same must be done for `loop` constructs with {`independent`,`gang`,`worker`,`vector`}. We do not show the OpenACC policy dispatch overload for brevity.