Automatic Mapping and Optimization to Kokkos with Polyhedral Compilation

Muthu Baskaran
Reservoir Labs
New York, NY
baskaran@reservoir.com

Charles Jin
Massachusetts Institute of Technology
Cambridge, MA
ccj@csail.mit.edu

Benoit Meister
Reservoir Labs
New York, NY
meister@reservoir.com

Jonathan Springer
Reservoir Labs
New York, NY
springer@reservoir.com

***Abstract*—In the post-Moore's Law era, the quest for exascale computing has resulted in diverse hardware architecture trends, including novel custom and/or specialized processors to accelerate the systems, asynchronous or self-timed computing cores, and near-memory computing architectures. To contend with such heterogeneous and complex hardware targets, there have been advanced software solutions in the form of new programming models and runtimes. However, using these advanced programming models poses productivity and performance portability challenges. This work takes a significant step towards addressing the performance, productivity, and performance portability challenges faced by the high-performance computing and exascale community. We present an automatic mapping and optimization framework that takes sequential code and automatically generates high-performance parallel code in Kokkos, a performance portable parallel programming model targeted for exascale computing. We demonstrate the productivity and performance benefits of optimized mapping to Kokkos using kernels from a critical application project on climate modeling, the Energy Exascale Earth System Model (E3SM) project. This work thus shows that automatic generation of Kokkos code enhances the productivity of application developers and enables them to fully utilize the benefits of a programming model such as Kokkos.**

***Index Terms*—Compiler, mapping, exascale programming models, Kokkos, E3SM application.**

I. Introduction

The technical landscape in high-performance computing (HPC) for achieving exascale is rapidly evolving, with a shift toward more diversity in processing solutions. The hardware architecture of modern supercomputers and future exascale systems is increasingly designed to include novel custom and/or specialized processors to accelerate the systems. Other hardware architecture trends include asynchronous or self-timed computing cores, advanced packaging, and near-memory computing architectures. As a consequence of this hardware evolution, the nature of processor hardware targets is becoming more heterogeneous and complex.

This shift in hardware processing solutions demands advanced software solutions in the form of either new programming models and runtimes or new and advanced features in established programming models. Such solutions would need to contend not only with the challenge of developing applications that can make efficient use of massive amounts of parallelism, heterogeneous nodes with specialized accelerators, and deep memory hierarchies, but also with concerns such as power constraints, diverging clock speeds, and resiliency against node failures.

Obtaining the best performance from new hardware will require application developers to use these new and advanced solutions in programming models and runtimes. However using these advanced programming solutions often requires complex programming and demands expensive and rare expertise. These challenges could be addressed by an automatic mapping and optimization tool, which could play a critical role in the HPC and exascale software stack in conjunction with the advanced parallel programming models and runtimes. Automatic mapping and optimization creates value by performing complex code generation for new and advanced programming models automatically and dependably, and thereby expands the user base of advanced software technologies in programming models. Automatic mapping and code generation can also significantly reduce the cost and effort of application rewriting or porting that constantly happens in the HPC domain.

Through this work, we take a significant step towards addressing the performance, productivity, and performance portability challenges faced by the HPC and exascale community by presenting an automatic mapping and optimization framework based on R-Stream, a robust auto-parallelizing polyhedral compiler [1]. The framework takes sequential code and automatically generates high-performance code in Kokkos [2], a performance portable parallel programming model targeted towards current and next generation computing systems. Further, we demonstrate the productivity and performance benefits of optimized mapping to Kokkos using kernels from Energy Exascale Earth System Model (E3SM) project, a critical application project on climate modeling. Specifically, we use kernels from High Order Method Modeling Environment (HOMME), the key atmospheric component of E3SM. Our evaluation results emphasize that automatic generation of Kokkos code enhances the productivity of application developers and enables them to fully utilize the benefits of a programming model such as Kokkos. Our contributions through this work are summarized as follows:

+ We develop and provide support for automatic mapping and code generation with advanced exascale-focused transformations to the Kokkos programming model.
+ We demonstrate performance and productivity benefits via kernels from a critical climate modeling application.

II. Background

A. *R-Stream Compiler*

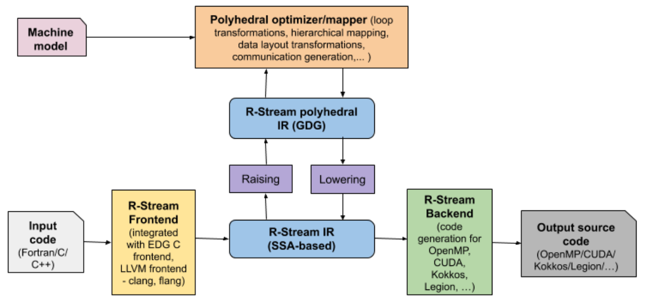
Machine model
Polyhedral optimizer/mapper (loop transformations, hierarchical mapping, data layout transformations, communication generation,...)
R-Stream polyhedral IR (GDG)
Raising
Lowering
Input code (Fortran/C/ C++)
R-Stream Frontend (integrated with EDG C frontend, LLVM frontend - clang, flang)
R-Stream IR (SSA-based)
R-Stream Backend (code generation for OpenMP, CUDA, Kokkos, Legion, …)
Output source code (OpenMP/CUDA/ Kokkos/Legion/…)

Fig. 1. R-Stream Compiler Workflow

R-Stream is a source-to-source optimizing compiler based on the polyhedral model. It takes in sequential C/C++/Fortran code as input and generates parallel code in the form of OpenMP, CUDA, OpenCL, Global Arrays, OCR, or Legion (depending on the target backend specification). In R-Stream, the programmer expresses what to do, not how to do it. R-Stream figures out mappings of the "what" into the "how." R-Stream has unique capabilities for high-level mapping of computations to parallel computing systems, including mapping to heterogeneous and deep hierarchical architectures. R-Stream includes (but is not limited to): optimized parallelism-locality-contiguity affine scheduling, hierarchical iteration space tiling, scratchpad memory management, communication generation within/across processing elements, energy-proportional scheduling, an auto-tuner ("ARCC") for tuning compiler mapping options, and ability to target multiple computing platforms (x86 multicores, GPUs, clusters of x86 multicores, specialized architectures) and multiple execution models [1], [3]–[6].

R-Stream is directed by a declaration of the target machine, expressed in a hierarchical machine-model description language. This enables R-Stream to seamlessly address architectural diversity and guide the optimizations based on the target architecture. High-level (polyhedral) optimizations in R-Stream operate with a polyhedral mapper representation called generalized dependence graph (GDG). Polyhedral optimizations take a GDG as input and generate a new GDG with additional or altered information. Low-level optimizations occur on a different SSA-based IR, after high-level transformations are applied. The process of converting from SSA-based IR to GDG is called "raising" and the reverse process is called "lowering." Fig. 1 shows the workflow of the R-Stream compiler.

B. *Kokkos*

Kokkos is a C++ based parallel programming model that provides a rich set of abstractions for parallel execution and data management. With separate abstractions for execution space, execution policy, execution patterns, data space, data layout and more, Kokkos comprehensively provides concepts for programming and achieving performance portability.

A parallel computation in Kokkos is characterized by its parallel pattern, execution policy, and computational body. The three common data parallel patterns supported in Kokkos are `parallel_for`, `parallel_reduce`, and `parallel_scan`. A computational body is specified in Kokkos through a functor (a function with data) or C++11 lambda (a concise representation that is auto generated into a functor by the compiler). Data is passed to a computational body through the functor's data members. The iteration space of a computational body is passed to a Kokkos parallel pattern through an execution policy parameter (a C++ class called `ExecPolicy`) that specifies iteration space along with other execution properties. More details on specific features and abstractions of Kokkos are presented in the next section.

III. Automatic Mapping to Kokkos

In this section, we describe the compiler support in R-Stream for mapping and generating high-performance code for the Kokkos programming model. The features can be broadly classified into three types: (1) support for hierarchical parallelism, (2) support for heterogeneous mapping, and (3) support for data management.

A. *Support for Hierarchical Parallelism*

Kokkos provides abstractions for hierarchical parallelism with shared memory semantics, in order to capture the hardware trend toward hierarchies at node level. Applications written using hierarchical parallelism are then mapped to hardware features depending on the architecture. Currently, Kokkos uses three nested levels of parallelism. The first level of parallelism is a league of thread teams. Within each thread team, the second level of parallelism is individual threads which can run concurrently, and the final level of parallelism is vectorized operations run by a single thread.

Here we describe how we automatically generate code that takes advantage of these Kokkos features for hierarchical parallelism, including parallel dispatch, shared memory, and synchronization.

Kokkos offers several methods of parallel dispatch. The dispatch is abstracted into several distinct components: the `ExecutionSpace` specifies where the computation is to occur (e.g., a CPU using OpenMP, or a GPU using CUDA); the `ExecutionPolicy` specifies how the computation is to be parallelized (e.g., the loop bounds); and finally, a lambda or functor provides what the computation is (e.g., the loop body). The `ExecutionPolicy` exposes multiple levels of shared-memory parallelism to the programmer. We chose to instantiate the `ExecutionPolicy` using the `TeamPolicy` as this API offers the richest support for hierarchical parallelism. `TeamPolicy` is templated on `ExecutionSpace`, which controls the ultimate target for the parallel dispatch. We discuss how this is used to achieve heterogeneous support in section III-B.

`TeamPolicy` allows the developer to set the three levels of hierarchy explicitly; any unspecified parameters are implicitly filled with `Kokkos::AUTO`, which lets Kokkos select a parameter based on the hardware of the target system. Because R-Stream already accepts as input a model of the target machine, we choose to generate a `TeamPolicy` with all three levels set explicitly. At a high level, given the target processor dimensions (e.g. a CPU grid, nodes in a NUMA machine, a CUDA GPU), and the dimensions of the computation, R-Stream transforms the computation space to expose outer levels of parallelism. The outer levels of parallelism are then mapped to the league, team, and vector dimensions, with heuristic tiling to coarsen the grain of parallelism. This also allows R-Stream to exploit resources in a hierarchical manner, such as coordination between threads in a team. In section III-B, we discuss how the determination of the hierarchy dimensions is specialized for different targets (i.e., OpenMP vs. CUDA).

Our support of Kokkos's abstractions for hierarchical parallelism enables explicit usage of coordination and shared resources between the units at the same level of hierarchy. This allows, for instance, threads in a team to use an explicitly-managed scratchpad memory and synchronization to execute more complex computations. R-Stream is able to target both of these types of hierarchical resources, and we implement support for automatically generating code for both team-based synchronization and scratchpad memories (which we describe in section III-C).

B. Support for Heterogeneous Mapping

Kokkos provides support for heterogeneous systems by providing abstractions for parallelism and data that are portable across different backend programming models, e.g. OpenMP and CUDA. At a high level, adapting code to different backends is as simple as changing the Kokkos `ExecutionSpace` and `MemorySpace` template parameters for computations and data, respectively. For instance, the following code creates and launches a kernel for parallel dispatch using 16 teams of 16 threads for OpenMP execution (changing the `ExecutionSpace` template parameter to `Kokkos::Cuda` does the same for CUDA execution):

```
Kokkos::parallel_for(Kokkos::TeamPolicy
<Kokkos::OpenMP>(16, 16),KOKKOS_LAMBDA( ... ));
```

`KOKKOS_LAMBDA` is a special macro that expands to different syntax depending on the target (e.g., it marks the lambda with the `__device__` keyword if the target is CUDA).

The analogous concept for data is a `View`, which provides abstractions that relieve the programmer of, most notably, explicit data movements and layout. For instance, a view could be allocated in the CUDA memory space to be accessed only by the GPU device using `Kokkos::CudaSpace`, in the "unified" memory space to be accessed by both the host and the device using `Kokkos::CudaUVMSpace`, or in the host memory space to be accessed only by the host using `Kokkos::HostSpace`.

We develop a method to generate code for both the `Kokkos::OpenMP` and `Kokkos::Cuda` backends. Our techniques leverage the common abstractions provided by Kokkos as much as possible, so that the polyhedral compiler flow can proceed down a unified mapping path. This is not only easier to maintain and extend from a development perspective, but also follows the Kokkos portability philosophy more closely than developing completely separate paths for each backend. Our support for different targets is accomplished by specializing this unified mapping path, which not only allows for code generating the different template arguments, but also some target-specific optimizations.

R-Stream supports a host + accelerator mapping path for CPU + GPU execution, including memory movements and kernel launches. Because the machine model for this mapping path is quite similar to the abstract Kokkos machine model (e.g. hierarchical parallelism, deep memory hierarchies), we were able to adapt existing components within R-Stream to map to Kokkos.

In particular, we choose to base the Kokkos mapping path on the existing host + accelerator mapping path targeting CUDA GPUs. This has several benefits. First, as described previously, the CUDA machine model offers both hierarchical parallelism and deep memory hierarchies that map naturally to the corresponding Kokkos concepts. Second, the host + accelerator divide naturally mirrors the boundary between the CPU and Kokkos parallel dispatch kernels. For instance, just as data must be copied between the host and device in the CUDA execution model, data movements across the Kokkos parallel dispatch boundary are performed explicitly in some cases. This allows us to leverage existing layout optimizations when possible.

Targeting different backends is accomplished via specialization of this unified mapping path. The specializations needed to adapt the mapping path to OpenMP and CUDA are quite lightweight, which is due to a combination of (1) the use of a machine model as input to R-Stream, which abstracts out many target-specific concerns (e.g., hardware specifics such as cache sizes and processor dimensions) and (2) the abstractions provided by Kokkos, which allows for expressing parallelism in a way that interoperates on a variety of systems.

For the OpenMP mapping path, we perform two specializations. The first specialization has to do with the generation of `TeamPolicy` for specifying the dimensions of the parallel dispatch. In OpenMP, there are two levels of hierarchy: the first is a league, which consists of certain number of teams; the second is the thread team, which is composed of a number of threads. In theory, the vector width constitutes the third level of this hierarchy; however, specifying a vector width does not spawn any additional threads in the OpenMP target, so we choose to ignore it for now. The number of teams in a league and number of threads in a team is then derived from the machine model provided to R-Stream heuristically given the processor grid dimensions. The second specialization has to do with the ability to use raw memory when mapping a function to a memory space that is accessible by both the host and (logical) accelerator. Because the "accelerator" is simply a CPU in the OpenMP target, there is no need to perform any copies between the host and accelerator. Thus, the mapped kernel can simply use the original pointers for data. We allow this specialization to be controlled by a compile-time flag.

For the CUDA mapping path, we perform only one specialization, which is the generation of `TeamPolicy`. Similar to OpenMP, the `TeamPolicy` specifies the league size and thread team size. However, unlike OpenMP, the third level of the hierarchy actually spawns additional threads. The Kokkos specification maps the league size to a one-dimensional grid, the team size to the y dimension of the thread block, and the vector length to the x dimension of the thread block. Thus, the dimensions of the kernel launch are derived from the corresponding machine model parameters for the GPU. Note that this difference in the Kokkos specification also prevents code generated for a GPU from being run on a CPU simply by changing the `ExecutionSpace` template parameter, as the number of threads generated will be different. If this functionality is desired, we also allow for setting the machine model to set the maximum x dimension of the thread block to 1, which guarantees that the third level of hierarchy is unused.

C. *Support for Data Management*

Kokkos's `View` abstraction is a container for data. Each `View` is templated on a `MemorySpace`, which specifies where the `View` is accessible. A `View` is also templated on a layout, which determines how the underlying data is actually stored relative to the access semantics. For instance, `LayoutRight` corresponds to the regular C-style row-major layout for arrays which is more appropriate for `Kokkos::OpenMP` (with improved locality), whereas `LayoutLeft` results in column-major layout that is better for `Kokkos::Cuda` (for memory coalescing). The layout parameter can be explicitly provided at `View` initialization, or Kokkos can infer a likely layout based on the `MemorySpace` of the `View`.

Kokkos `Views` come in two types: managed and unmanaged. A managed `View` lets the Kokkos runtime control the allocation, and thus layout and management, of the underlying memory. Conversely, an unmanaged `View` lets the programmer control the underlying memory, and is essentially just a wrapper around raw memory to provide Kokkos semantics for data access. Some memory must be unmanaged (e.g., scratchpad memories). We implement custom data layouts using both managed and unmanaged Views. Because this choice has performance and usability tradeoffs which may change based on the intended usage, we expose this to the end user as a compile time flag `KOKKOS_MANAGED_VIEWS` which is passed to R-Stream.

In order to support managed `Views`, we insert prologues and epilogues for copying to and from the input arrays. These are simple loops without any layout transformations; rather, we allow Kokkos to select the optimal layout for the underlying data based on the `MemorySpace`. Accessing the `View` inside the mapped code is thus able to make use of Kokkos's layout transformations transparently. We use `Kokkos::HostSpace` when targeting OpenMP, and `Kokkos::CudaUVMSpace` when targeting CUDA. In the case of CUDA, this allows the view to be accessible on both the host and device, with the Kokkos runtime performing the necessary copies between the host and device whenever the View is passed to a device function.

Since an unmanaged `View` is simply a wrapper around a raw pointer to memory, the Kokkos runtime must be provided with the layout of the existing region of memory in order for the `View` access semantics to be accurately translated into the underlying array access. As all the arrays are passed in as C arrays, this means that the layout is always `LayoutRight`. Additionally, the Kokkos runtime does not automatically provide data management for unmanaged `Views`. So for the CUDA target, we also generate copies between the host and device. This provides an opportunity for R-Stream to generate a more optimal data layout for the device memory before it is converted to a `View`. For CUDA, this means enabling memory coalescing so that the threads in a team access consecutive locations. R-Stream estimates a parametric data access pattern and then performs a linear transformation on the indices that achieves this pattern. The linear transformation is then applied to the copy operations in order to realign the device arrays for coalesced memory access.

R-Stream automatically identifies opportunities for using scratchpads and supports automatic creation of local arrays from global arrays, including determination of the optimal shape and size of the local arrays. We adapt these existing capabilities in R-Stream to the Kokkos backend and build support for targeting Kokkos's abstractions for hierarchical scratchpad memories. We support Kokkos's team-level scratchpad abstraction (`team_scratch`). Kokkos also offers a thread-private scratchpad (`thread_scratch`), but we leave that to future work.

The Kokkos abstraction for a scratchpad corresponds to different hardware features depending on the target system. For instance, if the ultimate target is CUDA, then anything allocated on the team scratchpad is equivalent to shared memory, which is an on-chip memory allocated per thread block. Shared memory is accessed by marking allocations with the `__shared__` keyword inside a device kernel.

R-Stream uses a heuristic to determine the use of scratchpad memories from the data footprint of computation tiles. For each kernel that is launched via parallel dispatch, we approximate the data access pattern parametrically. Portions of arrays that are either read or written repeatedly by multiple threads in the same team are then "promoted" to a scratchpad. This means that we first allocate a fresh chunk of scratchpad memory, and for portions that are read, we insert a prologue in the kernel that copies the portion of the original array to the scratchpad, while for portions which are written to, we insert an epilogue in the kernel that copies the scratchpad back to the original array. All accesses to the original array in the computational body of the kernel are then replaced with references to the scratchpad. Finally, we sum up the total footprint of all the scratchpad arrays created and pass it to the Kokkos runtime when creating the `TeamPolicy` (to set the scratchpad size using the `set_scratch_size` function).

Feature	Kokkos Abstractions	R-Stream Support
Heterogeneity	Execution spaces and memory spaces: `Kokkos::OpenMP`, `Kokkos::Cuda`, ... Execution Policies: `TeamPolicy`, `RangePolicy`, `MDRangePolicy`, ...	Automatic generation of `Kokkos::OpenMP` and `Kokkos::Cuda` execution spaces Automatic generation of `TeamPolicy` and `RangePolicy` execution policies
Hierarchical parallelism	Parallel dispatch constructs: `parallel_for`, `parallel_reduce`, and `parallel_scan` Different levels in `TeamPolicy`: `league` → `team` → `vector`	Automatic generation of `parallel_for` dispatch Automatic generation of different levels in `TeamPolicy` constructs
Data management	Managed and unmanaged `View` Different types of memory spaces: `CudaSpace`, `HostSpace`, `CudaUVMSpace`, ... Team-level scratchpad and barriers (`set_scratch_size`, `team_scratch`, `team_barrier`)	Automatic generation of managed and unmanaged `View` in `CudaSpace`, `HostSpace`, and `CudaUVMSpace` memory spaces Automatic creation and management of scratchpad and barriers

TABLE I
KOKKOS ABSTRACTIONS SUPPORTED BY AUTOMATIC MAPPING THROUGH R-STREAM

We also add support for the team barrier synchronization mechanism provided by Kokkos using `team_barrier`. Team barrier synchronization gives the programmer a way to coordinate between threads in order to design more complex worksharing algorithms. Similar to scratchpads, the ultimate implementation of the Kokkos team barrier abstraction depends on the target system. In CUDA, team barriers correspond to the `__syncthreads` block-level synchronization primitive, which makes a thread wait until all the other threads in the block have reached the same point. R-Stream inserts a team barrier (1) after each copy into a scratchpad and (2) before each copy out of a scratchpad.

Table I presents a summary of the major Kokkos abstractions that are necessary to provide key high-performance mapping and code generation capabilities through R-Stream.

IV. EVALUATION

To demonstrate the capabilities of our approach to provide (1) performance, (2) productivity, and (3) performance portability benefits in mapping HPC applications, we map and benchmark kernels in HOMMEXX [7]. HOMMEXX is an architecture-portable and high-performance implementation of HOMME (the atmospheric component of E3SM), and is part of an ongoing exascale application effort to transform the original Fortran codebase into a C++ with Kokkos codebase that achieves performance portability on exascale systems. We thus compare the single-node performance of hand-tuned Kokkos kernels inside HOMMEXX against equivalent, easy-to-write C kernels that are mapped through R-Stream.

HOMME is a simulation of the earth's atmosphere. This simulation is handled by imagining a grid of squares wrapping the earth's surface. The number of squares is called the number of elements, represented in our code by the constant `NUM_ELEM`. This is how the granularity of the simulation is represented. It can be thought of as the metric for the problem size. Within each element, the atmosphere is modeled at 72 levels, or heights from the earth's surface. We represent this in our code by the constant `NUM_LEV`. Within each element at a certain level, a certain number of points in space are modeled, specifically `NP` x `NP` points (`NP` is fixed at 4 by the application scientists). Several pieces of information about these points such as temperature and pressure is modeled over a series of time steps (i.e., to simulate time passing). At a high level, elements are distributed across MPI ranks (across nodes of a supercomputer) and each rank uses Kokkos kernels for its compute-heavy work. Specifically, each MPI rank implements the logic for modeling points in the atmosphere via four main functors which have operators that are dispatched in parallel as Kokkos kernels.

In our profiling of HOMMEXX, we found that "sphere operators" such as `laplace_simple`, `gradient_sphere`, `vlaplace_wk_contra`, and `vorticity_sphere` are called extensively and repetitively throughout the code. We also noticed that these operators were relatively small in terms of lines of code but dominant in terms of their contribution to the total execution time. The `laplace_simple` kernel, for example, takes up about 11% of the total time of HOMMEXX despite being only about 100 lines of code. For an application with about 60,000 lines of Fortran and 12,000 lines of C++ code, this is a very time-critical kernel. For further discussion on evaluation and demonstration of R-Stream mapping to Kokkos, we use the `laplace_simple` kernel. We also note that we checked the correctness of R-Stream generated Kokkos code of all mapped kernels by plugging them back into HOMMEXX and running the regression tests of HOMMEXX.

For our experiments, we use/generate and compare the following versions of the `laplace_simple` kernel:

1) HommeXX-Kokkos-{OpenMP/Cuda}: the extracted original code from HOMMEXX;
2) Baseline-{OpenMP/Cuda}: baseline parallel code;
3) R-Stream-{OpenMP/Cuda}: Direct OpenMP/CUDA code generated by R-Stream's OpenMP/CUDA backend;
4) R-Stream-Kokkos-{OpenMP/Cuda}: Kokkos::OpenMP / Kokkos::Cuda code generated by R-Stream's Kokkos backend (developed in this work).

Fig. 2 shows the performance of original and R-Stream-generated optimized OpenMP, CUDA, Kokkos::OpenMP, and Kokkos::Cuda versions of `laplace_simple` kernel. The OpenMP versions (direct OpenMP as well as Kokkos::OpenMP) generated by R-Stream shows 2x performance improvement over the original HommeXX-Kokkos-OpenMP version on a 32-core Intel multicore system. The CUDA versions (direct CUDA as well as Kokkos::Cuda) generated by R-Stream shows 10x performance improvement over the original HommeXX-Kokkos-Cuda version on a GeForce

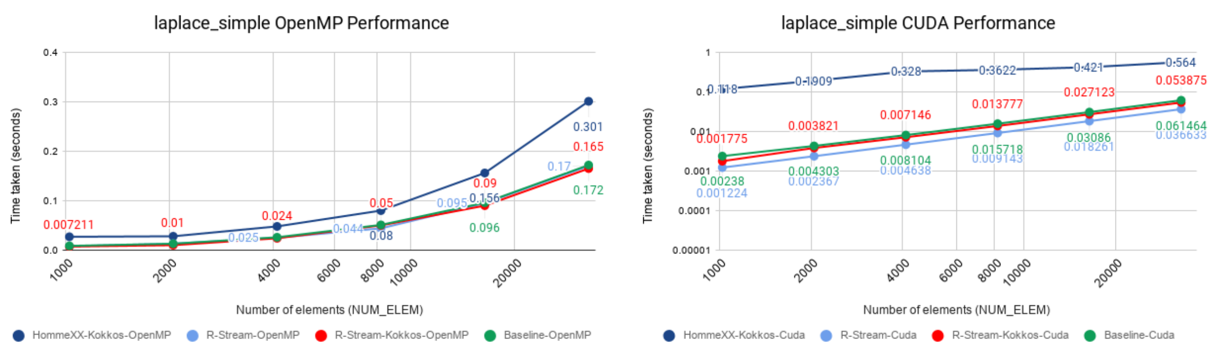

Fig. 2. Performance of original and R-Stream-generated optimized OpenMP, CUDA, Kokkos::OpenMP, and Kokkos::Cuda versions of laplace_simple kernel. Note that x-axis in both the graphs is in logarithmic scale and the y-axis in the CUDA graph is in logarithmic scale.

GTX 670 GPU that has 7 symmetric multiprocessors each with 192 computing cores. The direct OpenMP and CUDA versions generated by R-Stream perform slightly better than the corresponding R-Stream-generated Kokkos::OpenMP and Kokkos::Cuda versions. The generalizations in Kokkos for portability could be the reason for the difference in performance. We illustrate that R-Stream is able to automatically generate high-performance portable Kokkos::OpenMP and Kokkos::Cuda versions of `laplace_simple` that achieve high single-node performance. A key factor in achieving exascale performance is obtaining an optimized single-node performance. Hence the single node performance benefits shown by R-Stream validate the importance of the role of an optimization compiler such as R-Stream in the exascale software stack to achieve portable high performance on exascale systems.

The key optimizations from R-Stream that resulted in producing high-performance code versions for OpenMP, CUDA, and Kokkos are summarized below.

Hierarchical parallelism: Multi-level parallel loops are extracted and exploited for hierarchical parallelism. For Kokkos mapping, this is achieved by generating the levels of hierarchy provided by `TeamPolicy` in the form of league of thread teams, team of threads, and vectors in each thread. For direct OpenMP, we generate the `omp parallel for` directive for the appropriate parallel loops identified by the polyhedral mapper. For direct CUDA, we generate code that has thread block and thread dimensions associated with appropriate parallel loops identified by the polyhedral mapper.

Loop fusion for locality: The kernel has multiple loop nest instances of the form: `NP` $\rightarrow$ `NP` $\rightarrow$ `NUM_LEV`. The `NP` loops from different outer loops in the kernel are fused to exploit locality.

Memory coalescing for CUDA: This is a key optimization for direct CUDA and Kokkos::Cuda versions. For direct CUDA, we generate a cyclically distributed parallel thread-level loops enabling memory coalescing across the innermost CUDA thread dimension (`blockDim.x`). For Kokkos mapping, the vector level in the hierarchy is assigned appropriately to loops enabling memory coalescing.

Memory promotion: Reused array accesses are identified and promoted from slower memory (global DRAM) to faster memories (scratchpads and local registers). For Kokkos mapping, the promotion to scratchpad is achieved with team-level scratchpad and barrier functions (`team_scratch` and `team_barrier`). For direct CUDA, we generate `__shared__` arrays and local arrays, automatically copy in and out data, and synchronize using `__syncthreads`.

Communication hoisting: The promotion of memory accesses is hoisted to an outer level in the loop nest to enable more reuse of the promoted data.

The Kokkos constructs in the original Homme-Kokkos-{OpenMP/Cuda} versions also exploit the hierarchical parallelism in the kernel. However the loop fusion, memory promotion, and communication hoisting optimizations require code transformations that are non-trivial to perform manually. The non-trivial automatic code transformations in R-Stream-generated optimized Kokkos versions for OpenMP and CUDA not only contribute to higher performance but also require a 1.5-2x increase in the lines of code compared to the original Homme-Kokkos-{OpenMP/Cuda} versions. In conclusion, automatic generation of high-performance Kokkos code through R-Stream has been shown to be a more productive way to achieve higher performance portability.

V. Conclusions

Advanced software solutions in programming models and runtimes to address the evolving complexity and heterogeneity in modern and future computing systems pose productivity and performance portability challenges to programmers and application developers. Automatic mapping and optimization tools are a critical bridge between applications and underlying hardware targets for achieving high performance in a productive and portable way. In this work, we have developed an automatic mapping and optimization framework based on R-Stream, a robust auto-parallelizing polyhedral compiler, to enable the generation of high-performance code in Kokkos, a programming model targeted for current and next generation systems. We demonstrate the performance and productivity benefits from our framework using computation kernels from the E3SM climate modeling application. Our approach to automatic generation of Kokkos code thus enhances the productivity of application developers and enables them to fully utilize the benefits of a programming model such as Kokkos.

Acknowledgments

This material is based in part upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number DE-SC0019522. We would like to thank Divesh Otwani, a former employee at Reservoir Labs, who contributed to this work.

References

[1] B. Meister, N. Vasilache, D. Wohlford, M. M. Baskaran, A. Leung, and R. Lethin, "R-Stream compiler," in *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 1756–1765.

[2] 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*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[3] N. Vasilache, B. Meister, M. Baskaran, and R. Lethin, "Joint scheduling and layout optimization to enable multi-level vectorization," in *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, Paris, France, Jan. 2012.

[4] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin, "A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction," in *Third Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU-3*, Mar. 2010.

[5] M. Baskaran, B. Pradelle, B. Meister, A. Konstantinidis, and R. Lethin, "Automatic code generation and data management for an asynchronous task-based runtime," in *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*, Nov 2016, pp. 34–41.

[6] C. Jin, M. Baskaran, B. Meister, and J. Springer, "Automatic parallelization to asynchronous task-based runtimes through a generic runtime layer," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–11.

[7] L. Bertagna, M. Deakin, O. Guba, D. Sunderland, A. Bradley, I. Tezaur, M. Taylor, and A. Salinger, "Hommexx 1.0: A performance portable atmospheric dynamical core for the energy exascale earth system model," *Geoscientific Model Development Discussions*, pp. 1–23, 10 2018.