An Introduction to Kokkos

1st Yuan Luo
TUM School of Computation, Information and Technology
Technical University of Munich
Munich, Germany
yuan.luo@tum.de

***Abstract*—This paper presents a comprehensive evaluation of the performance and suitability of Kokkos, a parallel computing framework, for high-performance computing tasks. A matrix multiplication benchmark is used to assess the benefits, trade-offs, and scalability of Kokkos in comparison to OpenMP and CUDA implementations. Performance metrics including speedup, FLOPS, scalability, and memory efficiency are analyzed. Additionally, factors such as code readability, portability, compilation duration, and community support are considered. The evaluation reveals that Kokkos outperforms OpenMP in terms of execution time, but optimization efforts can reduce the performance gap. Compared to CUDA, Kokkos exhibits lower efficiency due to synchronization costs. Despite challenges in achieving performance parity with CUDA, optimization efforts improve scalability. Kokkos offers platform independence, but has longer compilation durations compared to OpenMP and CUDA. The evaluation provides insights into Kokkos' advantages, limitations, and potential trade-offs, enabling informed decisions for parallel computing tasks.**

***Index Terms*—Parallel computing, Kokkos, Performance evaluation, High-performance computing**

I. Introduction

Heterogeneous many-cores have become a crucial component of modern computing systems, ranging from embedded systems to supercomputers. To effectively utilize these heterogeneous systems, various programming models have been developed, categorized into low-level and high-level programming models [1]. Low-level models include CUDA, ROCm, Vivado C++, and DirectX, while high-level models encompass OpenMP, SYCL, and Boost. However, the abundance of programming models poses a challenge for programmers seeking to harness the power of different heterogeneous systems, as mastering each model can be demanding. Consequently, there is a need for a generic programming model capable of targeting diverse many-core architectures.

Kokkos [2] is a C++ programming model designed to address this challenge by providing a unified framework for developing performance-portable applications on major high-performance computing (HPC) platforms. It aims to abstract the complexities of different programming models and enable developers to write code that can efficiently execute on various many-core architectures.

In this work, we explore the fundamental concepts underlying Kokkos and shed light on how it achieves the abstraction of various programming models. We will examine the key features and design principles of Kokkos that enable it to target diverse hardware platforms, including CUDA, HIP, SYCL, HPX, OpenMP, and C++ threads. By understanding these concepts, developers can gain insights into the benefits and tradeoffs of using Kokkos compared to the original programming models. Through our analysis and discussion, we aim to provide a comprehensive overview of Kokkos as a versatile programming model for developing performance-portable applications on heterogeneous many-core architectures.

II. Background

Kokkos has been increasingly used in scientific computing. For example, Kokkos was used to optimize the QDP++ library for quantum chromodynamisc [3], accelerate astrophysics simulation [4], and simulate a stellar merger on the supercomputer Fugaku [5]. In addition, the well-regarded molecular dynamics simulation software, LAMMPS, is fully supported by Kokkos now [6]. Although Kokkos can contribute to scientific computation, it is better to know how much performance gain Kokkos can achieve and how much effort is needed to use Kokkos.

There are already some studies evaluating existing parallel programming models. [7] evaluates the performance and complexity of a wide range of parallel programming models, such as Kokkos, RAJA, and OpenACC, benchmarked by TeaLeaf. [8] compares the performance benchmarked by matrix multiplication of CUDA, OpenMP, and OpenACC on GPU only. These studies evaluate the programming models only in one aspect or on certain hardware. There is a need to evaluate them comprehensively under different benchmarks on different hardware.

III. Kokkos Programming Model

A program consists of execution and data. In order to deal with the difficulty of programming with various computing devices with various kinds of memory. Kokkos has built six basic core abstractions: Execution Spaces, Execution Patterns, and Execution Policies control parallel execution; while Memory Spaces, Memory Layouts, and Memory Traits control data storage and access [9] as shown in Figure 1. With these abstractions, developers can write general parallel program, instead of writing parallel program for a specific hardware explicitly. Kokkos will do the underlying work of mapping the general parallel program to the hardware in the way it considers best.

A. Execution Spaces

Heterogeneous computing has become attractive for parallel computing, as it can offer higher performance at lower

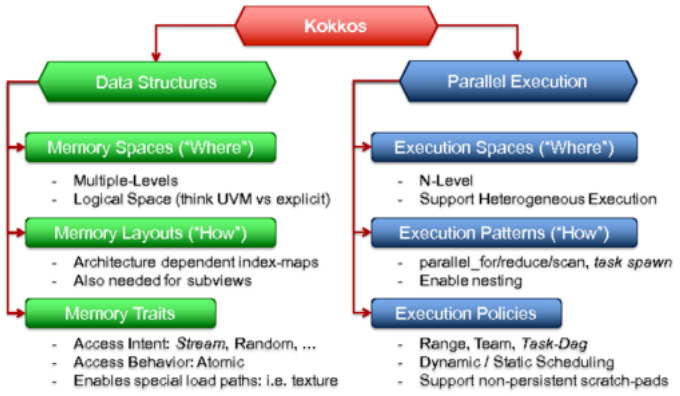

Fig. 1. The Core Abstractions of the Kokkos Programming Model (Taken from [10])

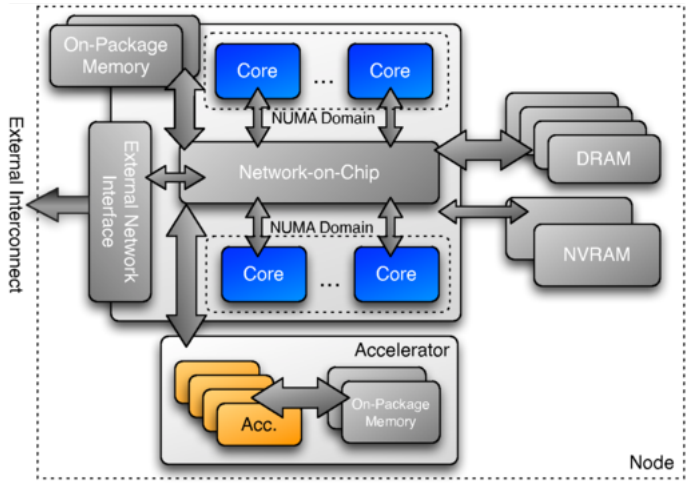

Fig. 2. Example Execution Spaces in a Future Computing Node (Taken from [10])

energy and/or cost consumption [11]. The Top 500 list of supercomputers [12] can prove this trend, as the architecture of computing node in most supercomputers are matched with the architecture shown in Figure 2. For example, PERLMUTTER consists of AMD CPU and NVIDIA GPU, FRONTIER is a hybrid system with both AMD CPU and AMD GPU, and SUMMIT is a combination of IBM CPU and NVIDIA GPU. The variety of computing resources offers the possibility to run an application more efficiently. However, it also makes it difficult for application developers to migrate the same application to different devices or architectures. To address this problem, the concept of Execution Space is introduced in Kokkos.

An Execution Space is a generic abstraction that represents an execution resource and an associated execution model [13]. It defines where the program is executed. For example, in a hybrid computing system containing CPU and GPU, there are can be three execution spaces. The CUDA streams on Nvidia GPUs are encapsulated by *Kokkos:Cuda*. For AMD GPU, it is *Kokkos::HIP*. *Kokkos::Serial* represents serial execution on CPU. While *Kokkos::Threads* stands for an execution space with a pool of threads. The program can be assigned to one or more Execution spaces at compile time, so to save the efforts of major rewriting for running the program on different devices.

The execution spaces in Kokkos can be organized into three categories:

- Host Serial: A Serial execution space with no parallelism or concurrency
- Host Parallel: Typically a threading model for CPUs, currently: OpenMP and Threads
- Device Parallel: Typically an attached GPU, currently: CUDA, OpenMPTarget, and HIP

Kokkos is also working on supporting more accelerating devices, such as FPGA and devices supported by SYCL.

B. Functor

Parallel computing can be classified as data parallelism and task parallelism. In data parallelism, a large data set is divided into smaller pieces. They are distributed among multiple processors or computing units. Each processor or computing unit operates on its own data based on the same processing logic. While in task parallelism, a large task in divided into sub-tasks which can be distributed among multiple processors or computing units and executed in parallel. Each task involves different operations on the same or different data. For instance, data parallelism is also commonly used in scientific simulations and data processing, while task parallelism is often used in parallel algorithms, such as sorting and searching.

No matter what kind of parallelism it is, we need to distribute the data and assign instructions on how to process the data to the processor or computing unit. There are two ways to distribute data and instructions. They can be distributed separately or together. In Message Passing Interface (MPI), data and operators are distributed independently: data are passed among nodes by API calls; operators are assigned to nodes based on their ranks. In parallel programming models such as OpenMP and Threading Building Blocks (TBB), data and operators are distributed together. Kokkos chose the same path as OpenMP and TBB. Computational bodies are given as *functors*, which is a function with data. The code snippet below shows a functor as an example. The data needed to calculate the atom force and the operator are both wrapped in a functor. The atom functor can then be executed in parallel. Sometimes it is inconvenient to build a functor for simple functionality. Kokkos has also introduced Kokkos lambda to address this issue, which is something similar to lambda in C++.

```
struct AtomForceFunctor {
   ForceType _atomForces;
   AtomDataType _atomData;

   AtomForceFunctor(atomForces, data) :
      _atomForces(atomForces) _atomData(data) {}
   void operator() (const size_t atomIndex) const {
      _atomForces[atomIndex] = calculateForce(
    _atomData);
```

```
    }
}

// how to call functor
AtomForceFunctor functor(atomForces, data);
Kokkos::parallel_for(numberOfAtoms, functor);
```

Listing 1. A Kokkos funtor

C. Execution Pattern

With Functor, the parallel program can be expressed as a group of independent subtasks. The further problem to solve is how these subtasks should be executed. In OpenMP, we can assign the parallel-for pragma with static or dynamic to instruct the execution order. In Kokkos, we have Execution Pattern. Execution Patterns are the fundamental parallel algorithms in which an application has to be expressed. There are three kinds of execution patterns in Kokkos:

- Parallel_for: execute a function in undetermined order a specified amount of times
- Parallel_reduce: which combines parallel_for() execution with a reduction operation
- Parallel_scan: which combines a parallel_for() operation with a prefix or postfix scan on output values of each operation

The code snippet below shows an example of `parallel_reduce` in Kokkos. It calculates the sum of all numbers in a matrix of *N* rows and *M* columns. The partial sum of each column is accumulated in `sum`, and they are added to `result` in the end. `parallel_reduce` can also be assigned with other reductions, such as `min` and `max`. It can save a lot of effort for programmers in some cases.

```
const size_t M = ...;
const size_t N = ...;
View<double**> A("a", M, N);
// ... fill A with some numbers ...
double result = 0.0;
Kokkos::parallel_reduce( N, KOKKOS_LAMBDA ( int j,
    double &sum ) {
    for ( int i = 0; i < M; i++) {
        sum += A(j, i);
    }
}, result);
```

Listing 2. Paralle_reduce in Kokkos

One thing to note is that it is not always better to use `Parallel_for`, because sometimes the overhead of creating parallel execution can outweigh the benefits when the loop count is really small. In such cases, Kokkos doesn't guarantee that `Parallel_for` will use all available parallelism. It usually runs in serial to be faster.

D. Execution Policies

An Execution Policy, in combination with an Execution Pattern, determines how a function is executed in Kokkos. One of the simplest forms of execution policies is the *Range Policy*, which is used to execute an operation for each element in a range. Kokkos also provides the *MDRangePolicy*, which defines an execution policy for a multidimensional iteration space. *TeamPolicy* will organize the threads into thread teams. Each team solves a portion of the overall problem.

In Kokkos, executing a parallel pattern with a range policy can be achieved by providing a single integer as the policy argument. Alternatively, an explicit *RangePolicy* object can be used. The code snippet below demonstrates these two equivalent approaches:

```
// These two calls are identical
parallel_for("Loop", N, functor);
parallel_for("Loop", RangePolicy<>(N), functor);
```

Listing 3. Range Policy in Kokkos

By default, the Kokkos library uses a range policy if a single integer is provided as the policy argument. This simplifies the syntax when working with simple loop iterations.

E. Memory Spaces

As shown in Figure 3, future high-performance computing nodes may have multiple types of memory available. To manage these memories effectively, Kokkos abstracts them into Memory Spaces. Each memory space provides a finite storage capacity for allocating and accessing data structures. Different memory space types have distinct characteristics regarding accessibility from execution spaces and performance considerations. For instance, GPU memory (e.g., GDDR) prioritizes high throughput, while CPU memory (e.g., DDR) focuses more on latency.

Kokkos provides appropriate abstractions for allocation routines and associated data management operations for each memory space type. These operations include memory allocation, release, returning memory for future use, and copy operations. The following example demonstrates copying data from host memory to device memory using Kokkos:

```
ViewMatrix x("x", M, N);
ViewMatrix y("y", N, M);
ViewMatrix z("z", M, M);

ViewMatrix::HostMirror h_x = Kokkos::
    create_mirror_view(x);
ViewMatrix::HostMirror h_y = Kokkos::
    create_mirror_view(y);
ViewMatrix::HostMirror h_z = Kokkos::
    create_mirror_view(z);

// Initialize x, y, and z with some numbers ...

// Copy the data from host memory to device memory
Kokkos::deep_copy(y, h_y);
Kokkos::deep_copy(x, h_x);
Kokkos::deep_copy(z, h_z);
```

Listing 4. Memory Space in Kokkos

F. Memory Layouts

Memory layouts in Kokkos express the mapping from logical or algorithmic indices to the address offsets for data allocations. The choice of memory layout can have a significant impact on the performance of data access patterns in algorithms. By adopting appropriate layouts for memory structures, an application can optimize memory access patterns

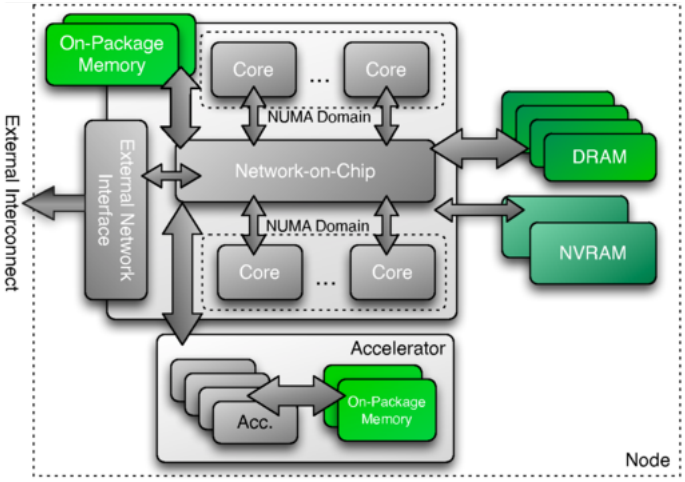

Fig. 3. Example Memory Spaces in a Future Computing Node (Taken from [10])

based on the specific requirements of the algorithm and the underlying hardware architecture.

In some programming languages, the language prescribes the memory layout. For example, arrays in FORTRAN are stored in column-major order, meaning that the elements of a column are stored adjacent to each other in memory. On the other hand, in languages like C, arrays are stored in row-major order, where the elements of a row are stored adjacent to each other in memory. The choice of memory layout in these languages is fixed and cannot be easily changed.

Kokkos provides flexibility in choosing memory layouts by supporting polymorphic layouts. This means that a data structure can be instantiated with different layouts at compile time or runtime, allowing for architecture-dependent optimizations. By selecting the appropriate memory layout, developers can optimize data access patterns and take advantage of specific hardware characteristics, such as cache locality or vectorization.

It's worth noting that choosing the right memory layout requires understanding the data access patterns and the characteristics of the underlying hardware architecture. Different algorithms may benefit from different layouts, and it often involves trade-offs between factors such as cache utilization, memory bandwidth, and vectorization efficiency.

G. Memory Traits

Memory Traits in Kokkos specify how a data structure is accessed in an algorithm. These traits provide information about the usage scenarios of the data, such as atomic access, random access, or streaming loads and stores. By assigning appropriate memory traits to data structures, an implementation of the Kokkos programming model can optimize load and store operations to achieve better performance.

The memory traits serve as hints to the compiler or runtime system about the intended access patterns of the data. This information can be used to guide code transformations and optimizations. For example, if a data structure is marked with the "random access" trait, the implementation may choose to optimize the memory access pattern for efficient random access operations. Similarly, if a data structure has the "atomic access" trait, the implementation can apply the necessary synchronization mechanisms to ensure correct atomic updates.

By leveraging memory traits, developers can provide additional information to the programming model implementation, enabling it to make better decisions regarding memory operations and optimizations. This can result in improved performance and efficiency, especially when working with complex algorithms or architectures with specific memory access requirements.

H. View

A View in Kokkos is a lightweight C++ class that represents an array or a multidimensional data structure. It consists of a pointer to the underlying data array and some metadata. Views provide a convenient and portable way to work with data in Kokkos, allowing for efficient memory access and manipulation.

In the code snippet below, a one-dimensional array and a two-dimensional array are declared using the View class. The size and shape of the arrays are specified in the constructor of the View. The memory access pattern of the View is determined at compile time based on the memory space or the memory traits being given, which enables memory performance portability across different devices.

```
typedef View<double*, Device> oneDimensionArray;
typedef View<double**, Device> twoDimensionArray;

oneDimensionArray("one", 10);
twoDimensionArray("two", 5, 5);
```

Listing 5. Kokkos View Usage

One of the challenges in efficiently utilizing different memory types on devices like Nvidia GPUs is mapping the data to the appropriate memory. In this case, memory traits can be given to a View to guide its memory behavior. For example, on CUDA devices, the "RandomRead" hint can be specified to utilize the texture cache for improved read performance.

Using Views in Kokkos allows developers to write device-agnostic code while achieving high memory performance. The compile-time mapping of data to memory and the availability of memory hints provide flexibility and efficiency when working with different devices and memory architectures.

IV. Evaluation

In this evaluation, we aim to assess the performance gains and trade-offs offered by Kokkos through a large-scale matrix multiplication task. Matrix multiplication is a computationally intensive operation commonly employed in machine learning and image processing applications. As such, it serves as an appropriate benchmark to evaluate the benefits and drawbacks of utilizing Kokkos. The Kokkos program is compiled into OpenMP and CUDA versions, which are then compared against the standard implementations using OpenMP and CUDA.

In addition to performance comparison, this evaluation also considers other essential metrics such as portability, code readability, and community support. These factors contribute to the overall assessment of Kokkos as a framework for parallel computing tasks.

The evaluation is conducted on a test platform consisting of two Intel Xeon Processor E5-2620 v4 CPUs, offering a total of 32 cores, and 128 GB of RAM. The system is further equipped with two Nvidia P100 graphics cards, each featuring 16 GB of GDDR memory. This hardware configuration provides an appropriate environment to examine the performance characteristics and scalability of Kokkos in relation to the chosen matrix multiplication task.

By conducting this evaluation, we aim to provide insights into the suitability of Kokkos for high-performance computing applications, shedding light on its advantages, limitations, and potential trade-offs in terms of performance, portability, code readability, and community support.

A. Performance Comparison with OpenMP

The performance study includes both non-optimized and optimized versions of the OpenMP and Kokkos applications, as well as a comparison between CUDA and Kokkos. In the non-optimized verions, all programs are parallelized in the very simple and basic way of that programming model.

Figure 4 depicts the findings, which highlight the absence of robust scalability in both the OpenMP and Kokkos algorithms. Notably, switching from a problem scale of 1250x1250 to 12500x12500 results in a considerable loss in kernel performance. However, Kokkos consistently outperforms OpenMP across all problem scales, especially at the lowest scale of 125x125, when Kokkos achieves an amazing performance that outperforms OpenMP by a factor of 11.5. While the performance advantage decreases as the problem scale increases, Kokkos still has a significant edge over OpenMP.

Kokkos regularly outperforms OpenMP in terms of execution time at all matrix scales. Even at the lowest scale of 125x125, where Kokkos trails OpenMP by a hair, the difference is insignificant. At bigger scales, Kokkos has much faster execution times than OpenMP.

The results of optimizing the OpenMP and Kokkos programs are depicted in Figure 5. The OpenMP application is optimized using matrix transposition and SIMD, whereas Kokkos is mainly optimized with the use of *Kokkos::TeamPolicy*. Both frameworks exhibit improved scalability after optimization, overcoming performance limitations encountered in non-optimized versions. However, in the optimized versions, Kokkos's performance advantage over OpenMP fades. At the smallest scale of 125x125, Kokkos outperforms OpenMP by 45.8%. At bigger scales of 1250x1250 and 12500x12500, Kokkos is even weaker than OpenMP by 59.9% and 56.6%, respectively.

The performance analysis of both non-optimized and optimized versions of the OpenMP and Kokkos programs reveals valuable insights. In the non-optimized scenario, Kokkos exhibits a significant performance advantage over OpenMP.

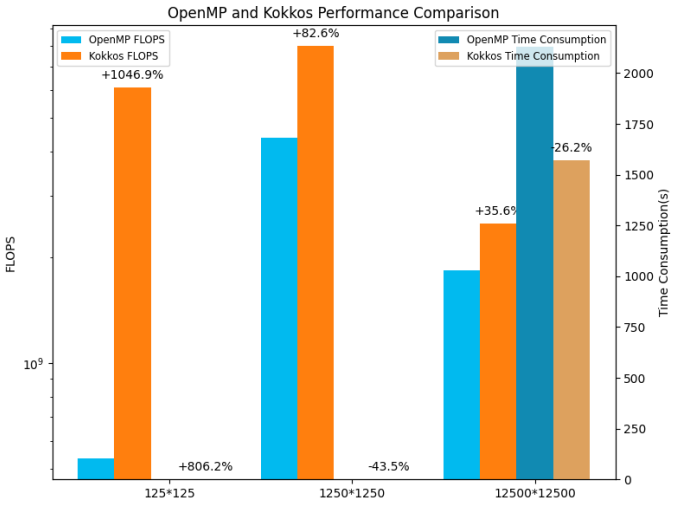

Fig. 4. OpenMP and Kokkos Performance Comparison (Non-optimized)

This advantage can be attributed to Kokkos' efficient data layout, task parallelism, and fine-grained parallelism. The optimized Kokkos program employs algorithmic optimizations, such as matrix transposition and SIMD operations, which narrow the performance gap with OpenMP. Additionally, the optimized Kokkos program leverages *Kokkos::TeamPolicy* to enhance team-based parallelism and achieve better scalability. However, despite the improved scalability, the performance advantage of Kokkos diminishes in the optimized versions. This can be attributed to the optimization efforts applied to the OpenMP program, which improve its efficiency and reduce the performance gap. Furthermore, it is worth noting that the optimized Kokkos implementation may not fully exploit the specific optimizations applied to the OpenMP version, resulting in a reduced performance advantage. Overall, the performance of Kokkos in both cases underscores the significance of efficient data layout, parallelism, and algorithmic optimizations in achieving superior performance and scalability.

B. Performance Comparison with CUDA

Figure 6 illustrates the performance comparison between the CUDA and Kokkos implementations. Both programs are evaluated in a non-optimized state, focusing on FLOPS (floating-point operations per second) and execution time as performance metrics.

For CUDA, the achieved FLOPS values are measured as 1.432×10^{12}, 1.139×10^{15}, 7.064×10^{17} for problem sizes of 125, 1250, and 12500, respectively. Conversely, the FLOPS values obtained for Kokkos are 1.256×10^{11}, 6.175×10^{13}, 2.990×10^{16} for the corresponding problem sizes.

The kernel performance of the Kokkos implementation consistently exhibits a lower level of efficiency compared to CUDA across all problem sizes. On average, Kokkos achieves approximately 10% of the FLOPS obtained by CUDA, indicating a significant performance disparity between the two implementations.

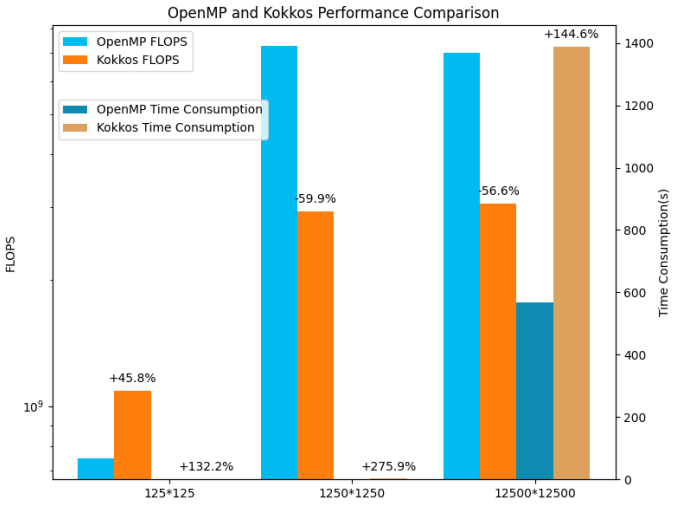

Fig. 5. OpenMP and Kokkos Performance Comparison (Optimized)

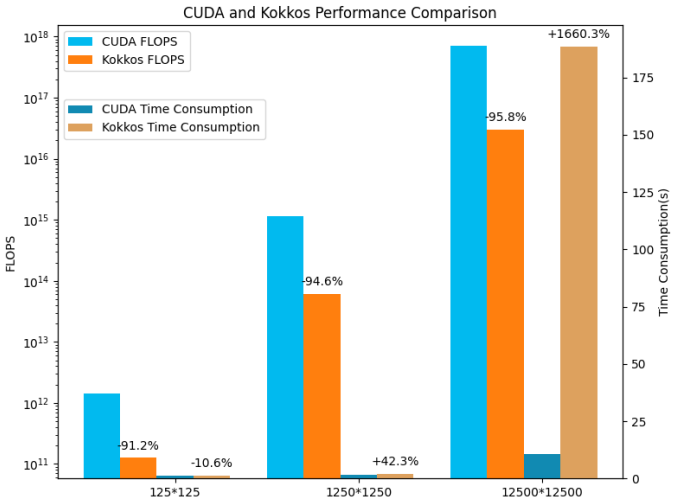

Fig. 6. CUDA and Kokkos Performance Comparison (Non-optimized)

In terms of execution time, Kokkos outperforms CUDA for the problem size of 125 by 125, achieving an execution time of 1.2334 seconds compared to CUDA's 1.379 seconds. However, as the problem size increases to 12500 by 12500, Kokkos experiences a substantial rise in execution time, taking approximately 188.292 seconds, whereas CUDA completes the computation in 10.6963 seconds.

The observed increase in execution time for Kokkos at larger problem sizes primarily stems from the finalization process. Kokkos necessitates synchronization and resource deallocation, resulting in additional overhead compared to the manual handling performed in the CUDA program. The generic and portable nature of Kokkos entails more time-consuming operations during finalization, leading to a considerable escalation in execution time.

Figure 7 presents the performance comparison between an optimized CUDA program and an optimized Kokkos program.

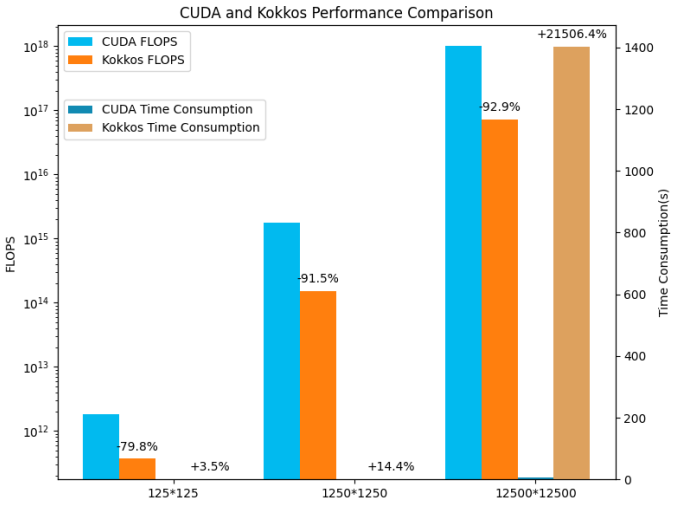

Fig. 7. CUDA and Kokkos Performance Comparison (Optimized)

In the optimized CUDA implementation, shared memory and tiling techniques are employed, while the Kokkos program remains the same as the previously optimized OpenMP version.

Both programs exhibit good scalability, maintaining a consistent increase in kernel performance as the matrix scale grows. The performance gap between the optimized CUDA and Kokkos programs is reduced compared to their non-optimized counterparts.

However, despite the optimization efforts, the total execution time of the Kokkos program still shows an overall increase, with a minor decrease observed at the matrix scale of 1250 by 1250. Notably, at the matrix scale of 12500 by 12500, the Kokkos program's execution time is now approximately 215 times higher than that of the optimized CUDA program. The Kokkos implementation requires more than 20 minutes to complete the entire program execution.

This significant increase in execution time can be attributed to the nature of the *Kokkos::TeamPolicy*, which divides threads into teams. This division introduces additional costs associated with synchronizing threads within teams and synchronizing the teams themselves.

The performance analysis reveals that, despite optimization efforts, the Kokkos program still lags behind the optimized CUDA program in terms of overall execution time. The use of *Kokkos::TeamPolicy* introduces additional synchronization costs, leading to a substantial increase in execution time, particularly at larger matrix scales.

C. Performance Comparison of Kokkos Program on Different Backends

The performance analysis of the optimized and non-optimized versions of the Kokkos program running on different backend programming models (OpenMP and CUDA) reveals interesting observations. In Figure 8, it is evident that the CUDA version exhibits improved kernel performance across all matrix scales, while the optimized Kokkos OpenMP pro-

gram experiences a significant performance drop at the scale of 125x125 and 1250x1250.

A closer examination of the parallelism granularity, as depicted in Figure 9, sheds light on the reasons behind these performance differences. The non-optimized Kokkos programs adopt a coarse-grained parallelism approach, where each thread calculates one row in the final matrix. Conversely, the optimized programs utilize a fine-grained parallelism strategy, where each thread computes one element in the final matrix. It is widely recognized that CUDA excels at fine-grained parallelism [14], which explains the performance gain observed in the optimized Kokkos CUDA program.

The performance drop in the optimized Kokkos OpenMP program at smaller matrix scales can be attributed to the overhead associated with frequent thread creation and destruction. The coarse-grained parallelism employed in these cases introduces additional costs, impacting the program's efficiency and resulting in a slowdown.

However, the performance gain of the optimized Kokkos OpenMP program at the larger matrix scale of 12500x12500 suggests that fine-grained parallelism, in this case, may offer better cache reusability compared to the coarse-grained approach. This finding highlights the significance of considering the specific problem characteristics and hardware architecture when selecting the appropriate parallelism granularity.

In conclusion, the analysis emphasizes that achieving desired performance with the Kokkos framework is highly dependent on optimizing for the targeted backend programming model. Different programming models possess distinct strengths and limitations, and tailoring the optimization strategies accordingly is crucial to leverage their advantages and mitigate potential performance bottlenecks.

The observations presented in this analysis contribute to a deeper understanding of the impact of parallelism granularity on the performance of the Kokkos program across different backend programming models. They underscore the importance of considering the specific characteristics of the problem and hardware architecture when optimizing for maximum performance.

D. Compilation Duration

When comparing the compilation duration of different programming models, such as OpenMP, Kokkos OpenMP, CUDA, and Kokkos CUDA, an interesting observation can be made. In general, the compilation time for OpenMP and CUDA programs is relatively short and not a significant concern. However, the situation changes when using Kokkos. Figure 10 highlights this difference. It is evident that Kokkos requires significantly more time for compilation compared to the standard OpenMP and CUDA frameworks. Specifically, Kokkos OpenMP has a compilation duration that is nearly 95 times longer than that of traditional OpenMP, while Kokkos CUDA takes approximately 58 times longer to compile compared to regular CUDA. It is important to note that these observations are based on a project with only one source file. Therefore, it is reasonable to anticipate that Kokkos would require even more time to compile for larger and more complex projects. Consequently, the compilation duration becomes a notable factor that cannot be overlooked when using the Kokkos programming model.

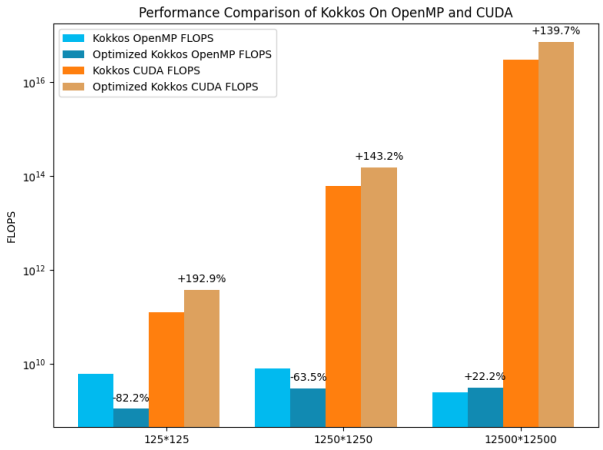

Fig. 8. Parallelism Granularity in Non-optimized and Optimized Programs

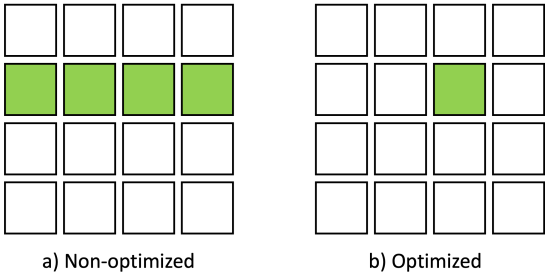

Fig. 9. Parallelism Granularity in Non-optimized and Optimized Programs

E. Code Readability and Portability

Code readability is a subjective concept, however OpenMP provides strong readability due to its simple pragma directives, which may be simply applied to existing code without requiring major adjustments. The pragmas are generally simple and intelligible, making it accessible to C/C++ developers. CUDA, on the other hand, as an API built to interact with C, C++, and Fortran, makes the transfer easier for developers with C/C++ knowledge. CUDA code is quite similar to C/C++ code, with the same grammar and structure. However, Kokkos, which is also written in C++, adds additional abstractions for execution and memory, making it significantly more difficult to read but still usable.

Kokkos excels in terms of portability because it prioritizes and encourages portability as a basic value. It strives to deliver excellent portability across many architectures and hardware platforms. Because OpenMP is a widely used parallel programming standard, it provides some portability. When vendor-specific extensions are employed, however, portability may suffer. CUDA, which was designed exclusively for Nvidia GPUs, has strong mobility inside the Nvidia GPU ecosystem.

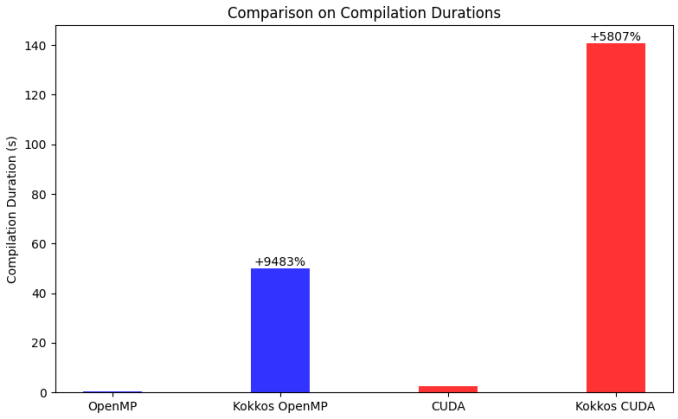

Fig. 10. Comparison of Compilation Duration

Extending beyond Nvidia GPUs, on the other hand, creates constraints and obstacles. While AMD's ROCm and HIP frameworks seek to support CUDA on AMD GPUs, there are still variances and limitations when compared to regular CUDA, limiting portability.

F. Community and Support

OpenMP has an active community with a dedicated website and forum for discussions and information sharing (www.openmp.org). NVIDIA's CUDA is supported via the NVIDIA Developer Program, which provides information and a developer forum (developer.nvidia.com). Kokkos has a burgeoning community focused on its GitHub repository (github.com/kokkos/kokkos), which offers access to source code, examples, and collaborative development initiatives. These communities provide significant resources and platforms for developers to interact with and learn from.

V. Summary and Future Work

This paper evaluates the performance and suitability of Kokkos, a parallel computing framework, through a comprehensive analysis using a matrix multiplication benchmark. The evaluation compares Kokkos with OpenMP and CUDA implementations and Kokkos running in different backends, considering performance metrics, code readability, portability, compilation duration, and community support. The findings reveal that Kokkos outperforms OpenMP in terms of execution time, but optimization efforts can narrow the performance gap. When compared to CUDA, Kokkos exhibits lower efficiency due to synchronization costs, but optimization improves scalability. Besides, when trying to achieve certain performance for a backend programming model, optimization targeting that programming model is required. Kokkos offers platform independence but has longer compilation durations. The evaluation provides valuable insights into Kokkos' advantages, limitations, and potential trade-offs, aiding in decision-making for parallel computing tasks.

For the future work, There are several points should be addressed. For example, what are the reasons for Kokkos taking so long to finalize the program when running in CUDA and how Kokkos' execution patterns are translated in different backend programming models can be studied. It will contribute to understanding Kokkos better and writing Kokkos programs with better performance across backend programming models.

Appendix

The source code for the project is available on GitHub at the following link:
`https://github.com/yyuan-luo/Kokkos-intro`

References

[1] J. Fang, C. Huang, T. Tang, and Z. Wang, "Parallel programming models for heterogeneous many-cores: a comprehensive survey," *CCF Transactions on High Performance Computing*, vol. 2, pp. 382–400, 2020.

[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.

[3] B. Joó, T. Kurth, M. A. Clark, J. Kim, C. R. Trott, D. Ibanez, D. Sunderland, and J. Deslippe, "Performance portability of a wilson dslash stencil operator mini-app using kokkos and sycl," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 14–25.

[4] P. Grete, J. C. Dolence, J. M. Miller, J. Brown, B. Ryan, A. Gaspar, F. Glines, S. Swaminarayan, J. Lippuner, C. J. Solomon, G. Shipman, C. Junghans, D. Holladay, J. M. Stone, and L. F. Roberts, "Parthenon—a performance portable block-structured adaptive mesh refinement framework," *International Journal of High Performance Computing Applications*, 12 2022.

[5] P. Diehl, G. Daiß, K. Huck, D. Marcello, S. Shiber, H. Kaiser, and D. Pflüger, "Simulating stellar merger using hpx/kokkos on a64fx on supercomputer fugaku," *arXiv preprint arXiv:2304.11002*, 2023.

[6] S. Moore, "The state of the lammps kokkos package." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2021.

[7] P. Czarnul, J. Proficz, and K. Drypczewski, "Survey of methodologies, approaches, and challenges in parallel programming using high-performance computing systems," *Scientific Programming*, vol. 2020, 2020.

[8] M. Khalilov and A. Timoveev, "Performance analysis of cuda, openacc and openmp programming models on tesla v100 gpu," in *Journal of Physics: Conference Series*, vol. 1740, no. 1. IOP Publishing, 2021, p. 012056.

[9] "Kokkos documentation," https://kokkos.github.io/kokkos-core-wiki/ProgrammingGuide/ProgrammingModel.html#programming-model, accessed: 2023-05-13.

[10] "Kokkos documentation," https://kokkos.github.io/kokkos-core-wiki/, accessed: 2023-05-28.

[11] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in heterogeneous computing," *Scientific Programming*, vol. 18, no. 1, pp. 1–33, 2010.

[12] "Top 500," https://www.top500.org/, accessed: 2023-05-16.

[13] C. R. Trott, D. Lebrun-Grandie, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez *et al.*, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2021.

[14] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.