An Introduction to Performing Parallel Programming with Kokkos

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, kernel execution time, scalability, and memory efficiency are analyzed. Additionally, factors such as code readability, portability, energy consumption, compilation duration, and community support are considered. The evaluation reveals that Kokkos outperforms OpenMP in terms of execution time, but the fine-grained programs can reduce the performance gap. Compared to CUDA, Kokkos exhibits lower efficiency due to the extra marco to call the standard CUDA functions. Despite challenges in achieving performance parity with CUDA, fine-grained parallelism improves scalability. 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 integral components of modern computing systems, spanning from embedded systems to supercomputers. To effectively harness the power of these heterogeneous systems, a variety of programming models have been developed, categorized into low-level and high-level approaches [1]. The low-level models encompass CUDA, ROCm, Vivado C++, and DirectX, while the high-level models include OpenMP, SYCL, and Boost. However, the proliferation of programming models presents a challenge for programmers who wish to leverage different heterogeneous systems, as mastering each model can be demanding. Therefore, there is a need for a generic programming model that can target diverse many-core architectures.

In response to this need, Kokkos [2] has emerged as a C++ programming model designed to provide a unified framework for developing performance-portable applications on major high-performance computing (HPC) platforms. The primary objective of Kokkos is to abstract the complexities associated with different programming models, enabling developers to write code that can efficiently execute on various many-core architectures.

In this work, our aim is to explore the fundamental concepts underlying Kokkos and elucidate how it achieves the abstraction of diverse programming models. We will delve into the key features and design principles of Kokkos, which enable it to target a wide range of hardware platforms, including CUDA, HIP, SYCL, HPX, OpenMP, and C++ threads. By comprehending these concepts, developers can gain valuable insights into the advantages and tradeoffs of using Kokkos compared to the standard programming models. Through our analysis and discussion, we seek to provide a comprehensive overview of Kokkos as a versatile programming model for developing performance-portable applications on heterogeneous many-core architectures.

II. Background

Since Kokkos was published in 2014, it has gained significant traction in the field of scientific computing, demonstrating its effectiveness in various applications. For instance, researchers have utilized Kokkos to optimize the QDP++ library for quantum chromodynamics, resulting in improved performance and computational efficiency [3]. Furthermore, Kokkos has been instrumental in accelerating astrophysics simulations, enabling scientists to explore complex phenomena with enhanced computational capabilities [4]. Notably, Kokkos has been leveraged to simulate a stellar merger on the supercomputer Fugaku, further attesting to its scalability and performance on cutting-edge architectures [5]. Additionally, the widely acclaimed molecular dynamics simulation software, LAMMPS, now offers comprehensive support for the Kokkos programming model, enabling researchers to leverage its benefits in their simulations [6].

While Kokkos has demonstrated its potential for scientific computation, it is crucial to assess the extent of its performance improvements and the level of effort required to incorporate Kokkos into existing workflows. Existing studies have evaluated various parallel programming models, providing insights into their performance and complexity. For example, Czarnul et al. conducted a comprehensive evaluation of parallel programming models, including Kokkos, RAJA, and OpenACC, using the TeaLeaf benchmark. However, this evaluation focused on specific aspects and did not provide a comprehensive comparison across different benchmarks and hardware configurations [7]. Similarly, Khalilov et al. compared the performance of CUDA, OpenMP, and OpenACC through matrix multiplication benchmarks on GPUs, but their evaluation was limited to a specific hardware platform [8].

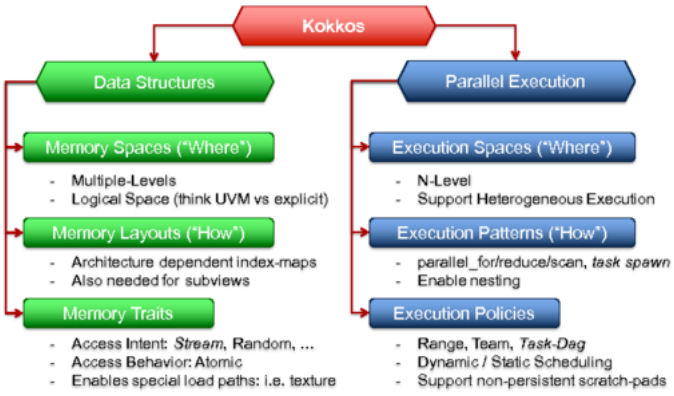

Fig. 1. The Core Abstractions of the Kokkos Programming Model (Taken from [9])

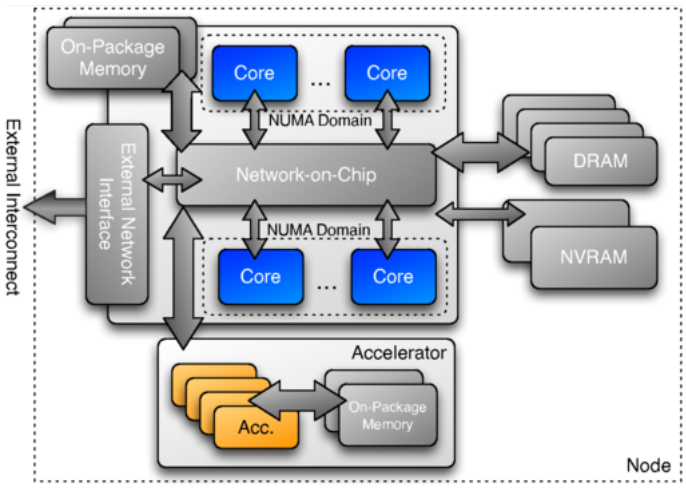

Fig. 2. Example Execution Spaces in a Future Computing Node (Taken from [9])

Hence, there is a pressing need for a comprehensive evaluation of these programming models, considering different benchmarks and hardware configurations.

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 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 is gaining popularity for parallel computing due to its ability to deliver higher performance while consuming less energy and cost [10]. This trend is evident in the architecture of supercomputers listed on the Top 500 [11], where computing nodes align with Figure 2. For instance, PERLMUTTER combines AMD CPU and NVIDIA GPU, FRONTIER integrates both AMD CPU and AMD GPU, and SUMMIT utilizes IBM CPU and NVIDIA GPU. This diversity of computing resources enables more efficient application execution. However, it poses challenges for developers when migrating applications across different devices or architectures. To address this issue, Kokkos introduces the concept of Execution Space.

An Execution Space is a generic abstraction that represents an execution resource and an associated execution model [12]. It defines where the program is executed. For example, in a hybrid computing system containing CPU and GPU, there can be multiple execution spaces. Kokkos provides encapsulation for CUDA streams on Nvidia GPUs using *Kokkos::Cuda*, and for AMD GPUs using *Kokkos::HIP*. *Kokkos::Serial* represents serial execution on CPU, while *Kokkos::Threads* represents a pool of CPU threads. By assigning different execution spaces to different parts of the program at compile time, Kokkos enables running the program on different devices without major rewriting efforts.

The execution spaces in Kokkos can be organized into three major 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

When implementing parallelism, data and instructions must be distributed to processors or computing units. There are two approaches for distributing data and instructions: separate distribution and combined distribution. In the Message Passing Interface (MPI), data and operators are distributed independently through API calls, with data passed among nodes and operators assigned based on node ranks. On the other hand, parallel programming models like OpenMP and Threading Building Blocks (TBB) distribute data and operators together. Kokkos follows the same approach as OpenMP and TBB. List 1 shows an example of a fucntor and how it is executed. The functor will be executed *numElements* times. Each time, the functor takes in an index offered by `Parallel_for`.

```
struct MyFunctor {
   DataType _data;
   MyFunctor(DataType data) : _data(data) {}
   void operator() (const size_t index) const {
      // Perform computations using _data and index
      // ...
   }
};
```

```
// How to call the functor
MyFunctor functor(data);
Kokkos::parallel_for(numElements, functor);
```

Listing 1. A Kokkos functor is declared with data, constructor, and operator. It is then initialized and gets called

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, there is something similar to that. Execution Patterns are the fundamental parallel algorithms in which an application has to be expressed in Kokkos. There are three kinds of execution patterns:

- `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 in listing 2 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`.

```
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. A Paralle_reduce example for calculating the sum of all the elements in a 2D array

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, it usually runs in serial to be faster, since Kokkos does not guarantee that `Parallel_for` will use all available parallelism.

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 in listing 3 demonstrates these two equivalent approaches. By default, the Kokkos library uses the *RangePolicy* if a single integer is provided. This simplifies the syntax when working with simple loop iterations. With *RangePolicy*, the specific execution space can be given as well.

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

Listing 3. The same RangePolicy in two different forms

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 low 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:

```
Kokkos::View<double**, Kokkos::Cuda> x("x", M, N);
Kokkos::View<double**, Kokkos::Cuda> y("y", N, M);
Kokkos::View<double**, Kokkos::Cuda> 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. Copy data from the host to CUDA devices 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 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

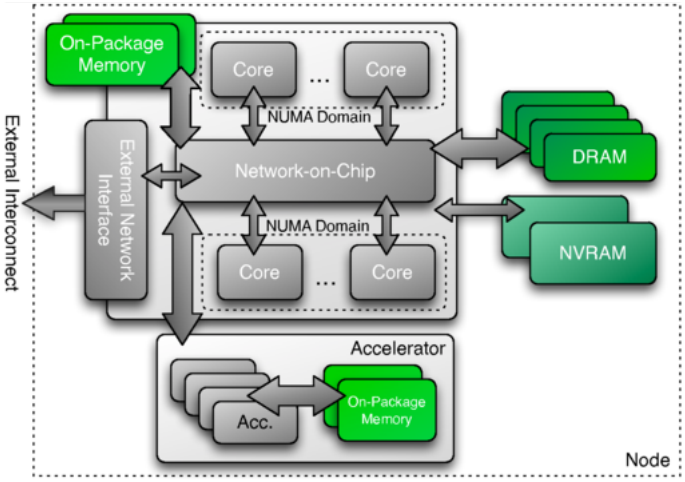

Fig. 3. Example Memory Spaces in a Future Computing Node (Taken from [9])

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.

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 in listing 5, 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. Define and initialize a 1D View and a 2D View in Kokkos

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, our objective is to analyze the performance improvements and trade-offs provided by Kokkos in the context of a large-scale matrix multiplication task. Matrix multiplication is a computationally demanding operation commonly utilized in various applications such as machine learning and image processing. Hence, it serves as a suitable benchmark to evaluate the advantages and disadvantages of employing Kokkos. We conducted a comparison between the Kokkos implementation compiled into programs using OpenMP and CUDA as backend models, and standard implementations using the OpenMP and CUDA programming languages. The programs, including Kokkos, CUDA, and OpenMP, were parallelized using both coarse-grained and fine-grained approaches, where calculations were performed at the row-level and element-level, respectively.

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.

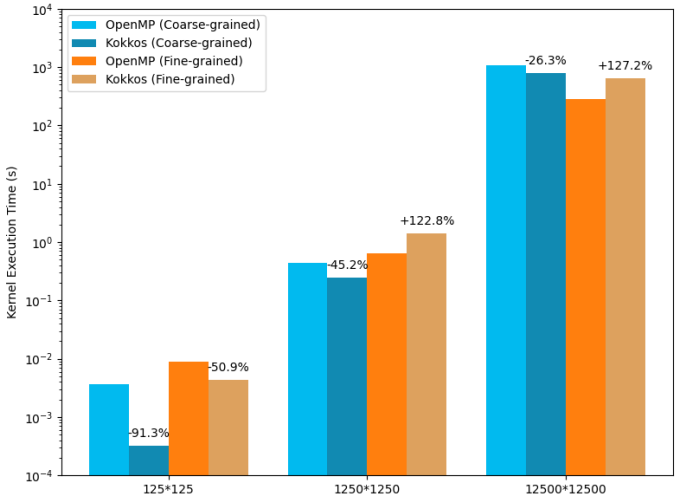

Fig. 4. Comparison of Kernel Execution Time for Coarse- and Fine-Grained Parallelism in Matrix Multiplication using OpenMP and Kokkos: Percentage indicates comparison to the corresponding left bar

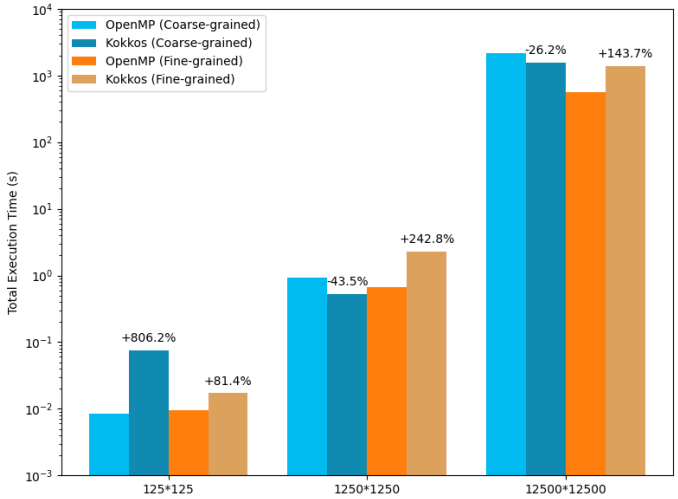

Fig. 5. Comparison of Total Execution Time for Coarse- and Fine-Grained Parallelism in Matrix Multiplication using OpenMP and Kokkos

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 one Nvidia P100 graphics card featuring 16 GB of HBM memory. The version of Kokkos, OpenMP, and CUDA for the evaluation are 4.0.0, 4.1.5 and 11.2 respectively. 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 the OpenMP and Kokkos applications with different parallelism granularity. Kokkos deploys fine-grained parallelism with the use of *Kokkos::TeamPolicy*.

Figure 4 illustrates the comparison of kernel execution time between the standard OpenMP and Kokkos programs for coarse- and fine-grained parallelism in matrix multiplication. The results reveal that the fine-grained programs exhibit better scalability compared to the coarse-grained programs. The kernel execution in fine-grained programs grows steadily as the size of the problem grows, while the coarse-grained programs suffer a significant relative increase in kernel execution time when transitioning from a problem scale of 1250 by 1250 to 12500 by 12500.

The analysis of both coarse-grained and fine-grained versions of the OpenMP and Kokkos programs provides valuable insights. In the coarse-grained scenario, Kokkos demonstrates a significant performance advantage over OpenMP, which can be attributed to its efficient data layout and task parallelism. On the other hand, the fine-grained Kokkos program leverages *Kokkos::TeamPolicy* to enhance team-based parallelism and achieve better scalability. However, despite improved scalability, the performance advantage of Kokkos diminishes in the fine-grained program. It appears that the Kokkos fine-grained parallelism introduces additional overheads that outweigh the performance gain from fine-grained parallelism. The additional might come from the additional synchronization require for a team of threads [13].

Figure 5 provides insights into the total execution time of the OpenMP and Kokkos programs. The fine-grained Kokkos program exhibits higher overall execution times compared to the fine-grained OpenMP program. This suggests that the fine-grained Kokkos program may have introduced additional overheads, impacting its execution performance. On the other hand, in the case of coarse-grained parallelism, the Kokkos program demonstrates shorter execution times compared to the OpenMP program at problem scales of 1250 by 1250 and 12500 by 12500. This observation further supports the notion that the fine-grained Kokkos program introduces additional overheads, as the coarse-grained Kokkos program performs more efficiently than its OpenMP counterpart.

B. Performance Comparison with CUDA

Figure 6 illustrates the execution time of the computation kernel of the CUDA and Kokkos implementations in both coarse- and fine-grained parallelism. In the fine-grained CUDA implementation, shared memory and tiling techniques are employed, while the Kokkos program remains the same as the previously fine-grained OpenMP version.

Both programs exhibit good scalability, consistently increasing kernel performance as the matrix scale grows. However, the standard CUDA programs keep outperforming Kokkos. In the coarse-grained programs, Kokkos program costs 10 times to 22 times more time than the standard CUDA program,

indicating a significant performance disparity between the two implementations. The performance gap between the fine-grained CUDA and Kokkos programs is reduced compared to their coarse-grained counterparts.

Figure 7 presents the total execution time comparison between the CUDA program and the Kokkos program. Kokkos only outperforms CUDA at the scale of 125 by 125 in coarse-grained parallelism. However, at larger scales, the Kokkos program incurs significantly higher execution times compared to the CUDA program. The observed increase in execution time for Kokkos at larger problem sizes primarily stems from the finalization process. For each execution space, it has its own implementation of the finalization. Kokkos requires 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. For example, in order to safely call standard CUDA functions to finalize the program, such as *cudaFreeHost* and *cudaEventDestroy*, Kokkos wraps them in a macro, *KOKKOS_IMPL_CUDA_SAFE_CALL*. This could explain why the finalization takes so long.

The execution time of the fine-grained Kokkos program still shows an overall increase compared to the coarse-grained program, 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 fine-grained 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.

C. Performance Comparison of Kokkos Program on Different Backends

The performance analysis of the fine-grained and coarse-grained versions of the Kokkos program on different backend programming models (OpenMP and CUDA) reveals distinct performance variations. By comparing the data presented in Figure 4 and Figure 6, it becomes clear that the same Kokkos program exhibits diverse performance characteristics when executed on different backend programming models.

This observation serves as a valuable guideline when utilizing Kokkos, emphasizing the importance of carrying out customized optimizations tailored to the specific backend model to attain the desired performance levels.

D. Energy Consumption

A new trend in high-performance computing is the increasing emphasis on environmental-friendliness, where energy consumption is a crucial factor to consider alongside performance. While the available data for evaluating the energy

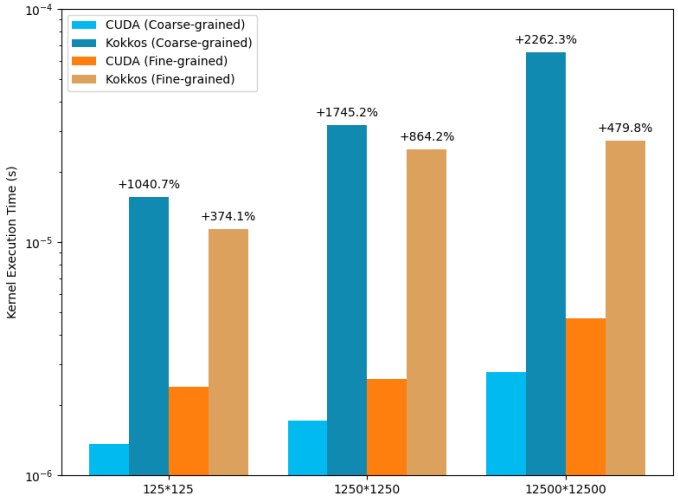

Fig. 6. Comparison of Kernel Execution Time for Coarse- and Fine-Grained Parallelism in Matrix Multiplication using CUDA and Kokkos

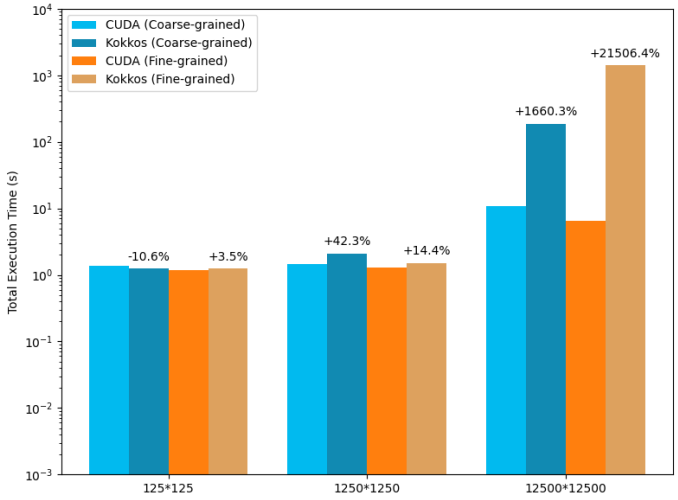

Fig. 7. Comparison of Total Execution Time for Coarse- and Fine-Grained Parallelism in Matrix Multiplication using CUDA and Kokkos

consumption of Kokkos and the standard program is limited due to the testing platform's constraints, some conclusions can still be drawn.

Both Kokkos and CUDA exhibit similar energy consumption during kernel computation. The peak power is estimated to be roughly 75 watts using the command *nvidia-smi*. However, the finalization process of the Kokkos program, as shown in Fig 7, consumes significantly more time at larger matrix scales. During the finalization process, the power stays around 35 wattage, which is close to the power of standby mode. However, the much extended finalization of Kokkos introduces a significant amount of additional energy consumption compared to the standard program. It's important to note that without specific data on CPU energy consumption and accurate GPU energy consumption, the evaluation of overall

energy efficiency between Kokkos and the standard program remains incomplete. However, the observation of increased energy consumption during the finalization process in Kokkos suggests the need for further investigation and optimization in this area.

E. Compilation Duration

When comparing the compilation duration with *Make* 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 8 highlights this difference, showing 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 18 times longer than that of traditional OpenMP, while Kokkos CUDA takes approximately 8 times longer to compile compared to regular CUDA. The increased compilation time can be attributed to the intricate process of mapping the Kokkos kernels onto various backend programming models and managing diverse memory layouts, as discussed in the previous section.

However, it is worth considering that the longer compilation duration of Kokkos may not pose a problem for larger projects. Kokkos encourages programmers who use Kokkos heavily in the project to compile with *CMake* [14]. *CMake* can help speed up the compilation process by taking advantage of features such as precompiled headers and incremental builds.

F. 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. A simple comparison can be found in the Appendix B.

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. 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.

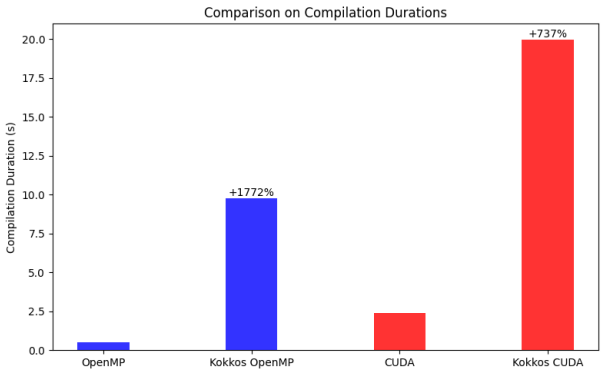

Fig. 8. Comparison of Compilation Duration with *Make*

G. 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 introduces key concepts in the Kokkos parallel computing framework and evaluates its performance and suitability using a matrix multiplication benchmark. The results demonstrate that Kokkos achieves superior execution time compared to OpenMP in coarse-grained programs, However, the advantage disapears in fine-grained programs. While Kokkos exhibits lower efficiency compared to CUDA due to factors like the macro used to ensure safe calls to standard CUDA functions, fine-grained parallelism enhances the scalability of Kokkos programs. Future work may involve investigating Kokkos' finalization time in CUDA, exploring its execution patterns across different backends, validating assumptions about compilation duration, and conducting more precise energy consumption comparisons. These endeavors aim to enhance our understanding of Kokkos and its capabilities.

Appendix A
GitHub Project

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

APPENDIX B
CODE READABILITY COMPARISON

A. Kokkos Code

```
#include <Kokkos_Core.hpp>

int main() {
    const int size = 1000;

    Kokkos::initialize();

    Kokkos::View<int*> a("A", size);
    Kokkos::View<int*> b("B", size);
    Kokkos::View<int*> c("C", size);

    Kokkos::parallel_for(size, KOKKOS_LAMBDA(const
    int i) {
        c(i) = a(i) + b(i);
    });

    Kokkos::finalize();

    return 0;
}
```

B. CUDA Code

```
#include <cuda_runtime.h>

global void vectorAdd(int* a, int* b, int* c, int
    size) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < size) {
        c[i] = a[i] + b[i];
    }
}

int main() {
    const int size = 1000;

    int* a, *b, *c;

    cudaMallocManaged(&a, size * sizeof(int));
    cudaMallocManaged(&b, size * sizeof(int));
    cudaMallocManaged(&c, size * sizeof(int));

    int blockSize = 256;
    int numBlocks = (size + blockSize - 1) /
    blockSize;

    vectorAdd<<<numBlocks, blockSize>>>(a, b, c,
    size);

    cudaDeviceSynchronize();

    cudaFree(a);
    cudaFree(b);
    cudaFree(c);

    return 0;
}
```

C. OpenMP Code

```
#include <omp.h>

int main() {
    const int size = 1000;

    int a[size], b[size], c[size];

    #pragma omp parallel for
    for (int i = 0; i < size; i++) {
        c[i] = a[i] + b[i];
    }

    return 0;
}
```

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/, accessed: 2023-05-28.

[10] 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.

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

[12] 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.

[13] "Kokkos documentation: Team barriers," https://kokkos.github.io/kokkos-core-wiki/ProgrammingGuide/HierarchicalParallelism.html#team-barriers, accessed: 2023-06-29.

[14] "Kokkos wiki: Compiling," https://github.com/kokkos/kokkos/wiki/Compiling/e4ac9a7b2e2f185d2088ff80d518b044dc3d7fb6#42-using-general-cmake-build-system, accessed: 2023-06-29.