Original scientific paper

A Comparative Study of OpenMP and Kokkos for CPU-Based Parallel Programming

Aleksa Paunović, Matija Dodović, and Marko Mišić

***Abstract* — The need for effective and portable parallel programming models has increased due to the complexity of high- performance computing systems, especially with the introduction of many-core processors like GPUs. Creating applications for various multi-core and many-core architectures frequently necessitates maintaining separate codebases, which increases development effort. In this paper, we evaluate Kokkos, a high- level C++ framework that allows for performance portability across architectures, by running five benchmark applications on a CPU and comparing their performance with OpenMP-based counterparts. The obtained results and corresponding observations are discussed in detail.**

***Keywords* — CPU-based parallelism, High-Performance Computing, Kokkos, OpenMP, Parallel Programming.**

I. Introduction

PARALEL architectures, consisting of multiple single-core processors connected via an interconnection network, has been prevalent in high-performance computing for decades [1]. Today, such systems are constructed by connecting multiple nodes, each of which may consist of, for example, two computing units. One computing unit is most often a multi-core general-purpose processor (CPU). The other typically serves as a computing accelerator, most often a Graphics Processing Unit (GPU) [2]. Fig. 1 shows one possible node design of such a system.

One of the challenges of parallel programming in general and the development of such systems in particular is code portability across a variety of computer platforms. This is particularly crucial when it comes to maintaining the same program's codebase and code performance across platforms. For example, the order in which data is accessed has a significant impact on program performance. Inefficient data-access patterns should be avoided to fully exploit the cache hierarchy of general-purpose CPUs. On GPUs and similar vector architectures, data access patterns should be standardised [4] [5]. In other words, the GPUs' memory accesses are the quickest when all threads access data sequentially.

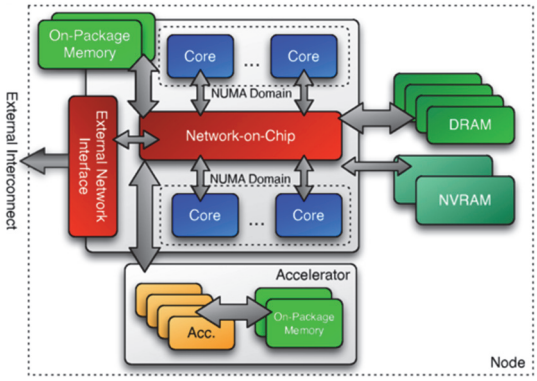

Fig. 1. A model of a high-performance computing node [3].

The portability of low-level frameworks is constrained [6] [7]. Some frameworks are low-level and vendor-specific, such as AMD ROCm/HIP and NVIDIA CUDA. Some, like OpenCL, provide an open standard for parallel programming across several heterogeneous platforms, although they don't have broader vendor support. Another open standard for parallel programming is OpenMP, which uses a directive-based methodology. OpenMP is compatible with CPU and GPU systems. OpenACC is a comparable GPU standard. As always, though, the programmer is responsible for controlling memory architecture and data access patterns. Recent work has also explored how large language models can assist in OpenMP- based code parallelization, offering insights into their current capabilities and limitations in aiding non-experts [8].

A higher-level paradigm called Kokkos C++ Performance Portability Ecosystem was created to deal with these issues [4]. Kokkos offers a collection of C++ programming language tools and libraries. Writing portable parallel programs with an emphasis on preserving performance across many architectures is its major purpose. The Kokkos Core library, which implements the most crucial data structures and capabilities for program parallelisation, is the foundation of this ecosystem [4], [9]. RAJA [10], a portability layer that allows C++ programs to exploit several programming paradigms with a single-source codebase, was developed using a similar concept. The SYCL standard [11], which is frequently utilised by the Intel oneAPI DPC++ implementation, is the other attempt for the C++ programming language.

The purpose of this article is to demonstrate the Kokkos library's capabilities and evaluate its CPU performance. For testing, five apps were chosen. These apps' sequential

Paper received June 2, 2025; revised November 3; accepted November 5, 2025. Date of publication: December 30, 2025. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Vujo Drndarević.

This work was financially supported by the Ministry of Science, Technological Development and Innovation of the Republic of Serbia under contract number: 451-03-137/2025-03/200103. The authors gratefully acknowledge the financial support.

Aleksa Paunović is with the School of Electrical Engineering, University of Belgrade, Serbia
(e-mail: pa243048m@student.etf.bg.ac.rs).
Corresponding author Matija Dodović is with the School of Electrical Engineering, University of Belgrade, Bulevar kralja Aleksandra 73, 11060 Belgrade, Serbia
(e-mail: mdodovic@etf.bg.ac.rs).
Marko Mišić is with the School of Electrical Engineering, University of Belgrade, Serbia (e-mail: marko.misic@etf.bg.ac.rs).

implementations and those built using the OpenMP parallel programming architecture were compared.

The paper is organized as follows. Section II describes the set of selected benchmark applications. Next, Section III describes the Kokkos framework with a focus on the Kokkos Core library. Section IV explains the implementation of the mentioned applications using the Kokkos Core library, while Section V presents the obtained results. The paper ends with a conclusion and directions for future work.

II. Benchmark Applications

The set of applications for parallelization was selected to test the various capabilities of the Kokkos framework and compare them with similar features of the OpenMP framework. The set of applications will be described in this chapter.

A total of five applications were chosen. Three of these applications were taken from the Parboil benchmark suite [12], which is used for performance testing. The remaining two applications were selected freely. The applications from the Parboil benchmark suite are: SGEMM (Single Precision General Matrix Multiply), TPACF (Two Point Angular Correlation Function), and SAD (Sum of Absolute Differences). The freely selected applications are a program that performs arithmetic calculations and a program that determines the number of prime numbers within a given range. These two applications are referred to as Arithmetic and Prime, respectively, in the project.

Table I briefly describes the selected applications. Additionally, apart from the descriptions of the applications themselves, two taxonomies were used for their classification. The first taxonomy, which describes the type of application, refers to the bottleneck of the core problem. If an application is memory-bound, the execution time largely depends on the time required for communication with the memory. On the other hand, if an application is compute-bound, the execution time primarily depends on the speed of the processor itself. Parallelization of such problems can often provide significant speedups [13].

The second classification describes the type of problem. This taxonomy, called the dwarf taxonomy, defines 13 “dwarfs”, each of which is meant to describe a type of problem that frequently occurs in parallel computing. The taxonomy is thoroughly described in [1], and here, we will focus on those "dwarfs" that describe the problems selected for parallelization:

Papers for the review process are submitted electronically online with the web-server interface.

1) **Dense Linear Algebra** problems deal with densely populated vectors and matrices (where most elements are not zeros). Access to the elements occurs in regular steps. For example, in matrices, this can be row-wise or column- wise access [1]. A subset of these problems is defined by the **BLAS** (Basic Linear Algebra Subprograms) standard [14].
2) The **N-body methods** classification refers to problems that calculate interactions between a large number of points. After the calculations, synchronization typically follows between all parallel units, whether they are threads on a GPU, different cores on a general-purpose CPU, or separate processors [15].
3) The execution of **MapReduce** problems is divided into two phases. The first, the **Map** phase, involves executing an independent function on each parallel unit. The function is considered independent because it does not require any synchronization between processes. Then, in the **Reduce** phase, communication occurs between all processes. During this communication, the set of results obtained from the Map phase is combined into a smaller set of results or a single result [1] [15].
4) In **Structured Grids** problems, data is organized in multidimensional grids. Spatial locality during data access is good and is defined by the algorithm itself. This type of problem is common in image and video processing [1] [15].

All implementations were obtained from [7], and how the implementations are modified will be explained in Section IV. Our running example will be the SGEMM application. Single Precision General Matrix Multiply is one of the most frequently studied applications for performance testing. This application involves the multiplication of two arbitrary matrices whose data consists of single-precision floating-point numbers [12]. The matrices are represented in input files as one-dimensional arrays and are organized by columns. The sequential implementation assumes that the second matrix is transposed in the file, so both matrices are accessed column by column, as shown in Listing 1 SGEMM is part of the BLAS standard [14], and according to the dwarf taxonomy, this problem belongs to the Dense Linear Algebra class of problems.

```
1  for (int mm = 0; mm < m; ++mm) {
2  for (int nn = 0; nn < n; ++nn) {
3          float c = 0.0f;
4     for (int i = 0; i < k; ++i) {
5        float a = A[mm + i * lda];
6        float b = B[nn + i * ldb];
7            c += a * b;
8         }
9         C[mm + nn * ldc] += c;
10     }
11  }
```

Listing 1. Kernel of the SGEMM application.

In the code, *A* denotes the first matrix, and *B* the second, while *C* is the result of the multiplication. To maintain the previously mentioned column-wise access, the variables *lda*, *ldb*, and *ldc* represent the stride.

III. Kokkos Framework

Kokkos consists of multiple libraries aimed at addressing different aspects of parallel application development [2]. Fig. 2 shows the entire Kokkos ecosystem. However, the three main components of this system are: Kokkos Core, Kokkos Kernels, and Kokkos Tools.

The Kokkos Core library defines a programming model for shared memory systems, providing abstractions for commonly used patterns in parallel programming to ensure code portability and performance across different platforms. One of the key features is the ability to specify an Execution Space during compilation, which determines

Table I:
SELECTED BENCHMARK APPLICATIONS AND THEIR CLASSIFICATION: DESCRIPTION, TYPE, AND DWARF TAXONOMY.

Application	**Description**	**Type** (Memory/Compute)	**Dwarf**
TPACF	Two-Point Angular Correlation Function, statistical analysis of body distribution in space	Memory	N-body
SGEMM	Single Precision General Matrix Multiply, matrix multiplication	Memory	Dense Linear Algebra
SAD	Sum of Absolute Differences, absolute difference between two images	Memory	Structured Grids
Arithmetic	Arithmetic number calculation in a given range	Compute	MapReduce
Prime	Determining prime numbers in a given range	Compute	MapReduce

whether a parallel kernel will execute on a CPU or a computing accelerator like a GPU, using models such as OpenMP, CUDA, or standard C++ threads. Additionally, for each execution space, Kokkos defines a corresponding Memory Space, which specifies where data resides during execution and its layout in structures such as Views. Other important abstractions provided by Kokkos include Memory Layouts, Execution Patterns, Memory Traits, and Execution Policies, which enable programmers to write code once and efficiently execute it across all supported platforms.

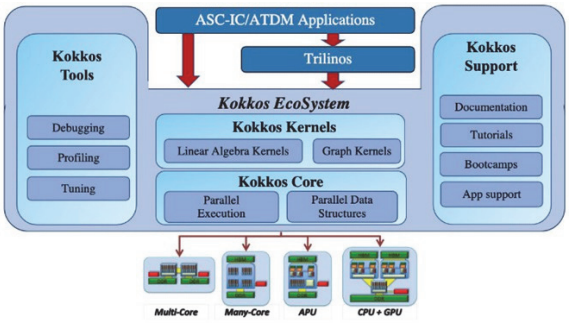

Fig. 2. Kokkos C++ Performance Portability Ecosystem [16].

These six core abstractions: Memory Spaces, Execution Spaces, Memory Layouts, Execution Patterns, Memory Traits, and Execution Policies—are illustrated in Fig. 3. Throughout this work, all C++ classes and functions will be referred to within the Kokkos namespace, and the complete interface is provided in Kokkos_Core.hpp. An example of a Kokkos kernel and instructions for compiling the library will also be provided later to demonstrate its practical usage.

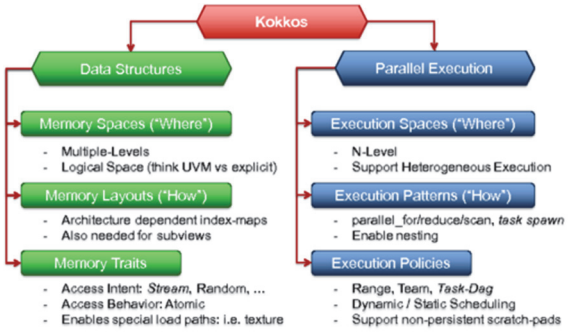

Fig. 3. Core abstractions in the Kokkos C++ Performance Portability Ecosystem [16].

The Kokkos Kernels library provides an implementation of the BLAS standard and certain kernels frequently used in graph- related problems. Besides the basic implementation of these kernels, which utilizes the Kokkos Core library, the library also applies certain optimizations specific to the architecture for which the written code is being compiled [4] [9].

Kokkos Tools is a set of tools for debugging and performance measurement of programs. To achieve this, the tools use the interface of the Kokkos Core library. Some of the capabilities of these tools include, for example, time measurement, memory usage analysis, and tracking memory events such as allocation and deallocation [4] [9].

IV. IMPLEMENTATION DETAILS

In Section II, the structure of the application kernels selected for parallelization was explained, as well as the classification of these applications according to various commonly used tax- onomies. This section explains the project structure and the dif- ferences between the downloaded and modified code provided in this work. Additionally, it will explain the implementation of parallel kernels using the Kokkos library, as well as those implemented using the OpenMP parallel programming model. The source code of the implemented benchmark applications is available at https://github.com/paunovicaleksa/diplomski.

Before moving on to the parallel implementations of the applications, it is important to comment on the changes made to the sequential code, which served as the basis for parallelization. In all applications taken from the Parboil benchmark suite, the method for parsing command-line arguments was changed, as well as the syntax of these applications. The original parsing method used functions and data structures found in the *parboil.h* header, while for the purposes of this project, the *getopt.h* header was used instead. The modification of the original code will be presented on the earlier-mentioned running example SGEMM.

The implementation of the SGEMM application, as discussed in Section II, was adapted for parallelization using the Kokkos library. The parallelization follows the structure of the original sequential kernel but utilizes the Kokkos views and lambda expressions to manage the parallel execution. The kernel uses the *MDRangePolicy* to merge and parallelize the two main loops, ensuring that memory access remains consistent with the original implementation. The complete Kokkos implementation of the SGEMM kernel is shown in Listing 2, while further comparisons with the OpenMP implementation are discussed later in this chapter.

```
Kokkos::parallel_for(Kokkos::MDRangePolicy({0, 0}, {m, n}),
    KOKKOS_LAMBDA (int mm, int nn) {
      float c = 0.0f;
      for (int i = 0; i < k; ++i) {
         float a = A(i, mm);
         float b = B(i, nn);
         c += a * b;
      }
      C(nn, mm) = C(nn, mm) * beta + alpha * c;
});
Kokkos::fence();
```

Listing 2. Kokkos Implementation of SGEMM Kernel.

For comparison, the OpenMP implementation of the SGEMM application is shown in Listing 3. This version uses the *collapse(2)* directive to parallelize the two outer loops and explicitly manages shared variables to ensure thread safety. The parallel kernel closely mirrors the structure of the sequential version, with loop collapsing improving load balancing across threads. While conceptually similar to the Kokkos version, the OpenMP directive results in a different distribution of work, which, as discussed later, significantly affects performance.

```
#pragma omp parallel for default(none)
    shared(m, n, k, A, B, C, beta, alpha,
    ldc, lda, ldb) collapse(2)
for (int mm = 0; mm < m; ++mm) {
  for (int nn = 0; nn < n; ++nn) {
    float c = 0.0f;
    for (int i = 0; i < k; ++i) {
      float a = A[mm + i * lda];
      float b = B[nn + i * ldb];
      c += a * b;
    }
    C[mm + nn * ldc] = C[mm + nn * ldc] * beta
        + alpha * c;
  }
}
```

Listing 3. OpenMP Implementation of SGEMM Kernel.

V. Results and Discussion

The selected applications were evaluated on an Intel(R) Core(TM) i7-11700F general-purpose processor. This processor features eight cores, each supporting two threads, with a total memory size of 64 GB. The cache hierarchy consists of three levels: the first level cache is split between instruction and data caches, sized at 256 KiB and 384 KiB respectively; the second level cache is 4 MiB; and the third level cache, shared among all cores, is 16 MiB.

Kokkos Core was set to use OpenMP as the underlying parallel framework. To ensure consistent time measurement, we utilized the timing functions from the parboil.h library, avoiding discrepancies that could arise from using the timers provided by the different parallel frameworks. Each application operates on its own dataset: the datasets for the Arithmetic and Prime applications are explicitly described, while datasets for the other applications can be found in [17].

Speedup is defined as the ratio between the execution time of the baseline, sequential implementation, and parallel implementation. For each application, a graph is presented to show the speedup compared to the sequential implementation, which is located in the CPU directory for each application.

In the following subsections, all results for each application are presented, including performance metrics, speedups, and a discussion of relevant observations.

A. SGEMM

SGEMM application was tested with two datasets, small and medium. Fig. 4 shows the speedup graph for this application. In the case of SGEMM, OpenMP demonstrates good results, even though the memory access pattern is not optimal. The Kokkos implementation, although faster than the sequential implementation, is nearly twice as slow as the OpenMP implementation. This difference cannot be explained solely by the overhead of the Kokkos Core library or by the time required to access a *Kokkos::View*. The discrepancy in execution time is due to the different behavior of the *MDRangePolicy* execution policy in Kokkos compared to the collapse directive in OpenMP. This difference will be further explained in the discussion.

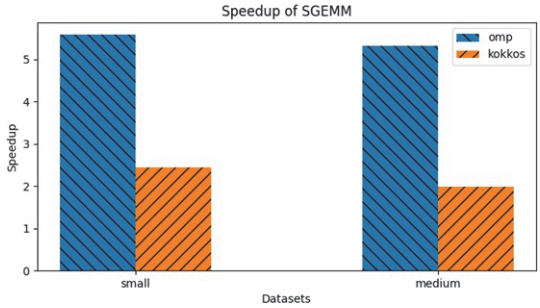

Fig. 4. SGEMM application speedup.

B. TPACF

The TPACF application was tested using three datasets. The speedup graph is shown in Fig. 5. Each solution provides similar performance, with the Kokkos solutions being slower due to the overhead of the library. In any case, a slowdown compared to the sequential solution is noticeable for each implementation. The cause of this slowdown is the cost of atomic access to the array elements. The histogram values, i.e., the values of the *data_bins* array elements, are not evenly distributed. A large number of distances between bodies fall into the last few entries of the histogram. Since the threads spend most of their time accessing the same elements, they must wait for atomic writes by other threads before they can write to the array themselves.

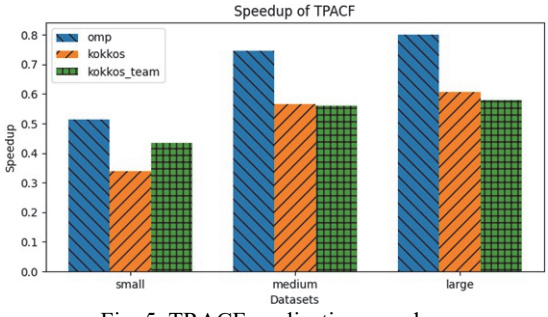

Fig. 5. TRACF application speedup.

Since the *data_bins* array is only written to during the kernel's execution, it is possible to create a solution where each thread has its own local array to write into. At the end of the kernel's execution, all local arrays would need to be merged into one. This solution would use significantly more

memory, especially for a large number of threads. Although this solution would avoid the atomic access problem, other issues might arise. In any case, this solution was not implemented.

C. *SAD*

The SAD application was tested on two datasets. Fig. 6 shows the speedup graph for this application. The OpenMP implementation achieves modest speedups in both cases. On the other hand, the Kokkos implementation results in a slowdown compared to the sequential implementation in both cases. As shown in the previous chapter, the Kokkos implementation uses the *MDRangePolicy* with a depth of 4. It is possible to implement it with a depth of up to 6. However, changing the policy depth did not improve performance in any of the cases. This holds true for a depth of 2 as well, which is equivalent to the depth of the OpenMP solution. Different loop distribution strategies also had no significant impact on performance, nor did changes to block sizes or the iteration order. The impact of calling *Kokkos::subview()* on the performance of this implementation is negligible, as no data is copied in memory.

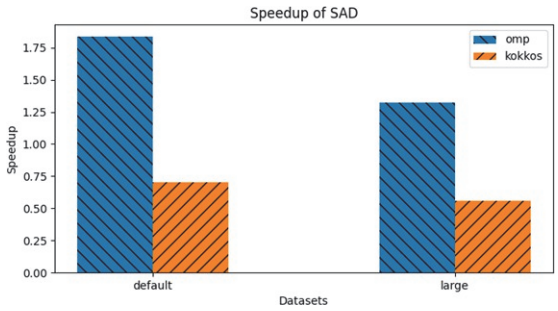

Fig. 6. SAD application speedup.

As with the SGEMM application, the performance difference stems from how the OpenMP *collapse* directive and the Kokkos *MDRangePolicy* distribute work. More details on this will be discussed in the following subsection.

D. *Aritmetic*

The Arithmetic application was also tested with two datasets, one with a search range of n=10 000 and the other with n=10 000 000. For the smaller search range, the worst results were observed with the Kokkos task-based implementation. This is expected, as the task-based implementation has a slightly higher overhead compared to others. For this dataset, the standard Kokkos Core implementation produced the best results. Implementations for small values of the input range n essentially behave similarly to the sequential implementation. For extremely small input ranges, the desired number of arithmetic numbers will be exceeded in the first iteration.

For large n, all implementations achieved similar speedups. The performance differences are negligible and can be attributed to measurement errors. Nevertheless, the speedup is significant for every implementation. This result is expected for this type of problem, as thread synchronization is minimized, and the bottleneck is not memory access but the computation performed by the processor. Fig. 7 shows the speedup graph for this application.

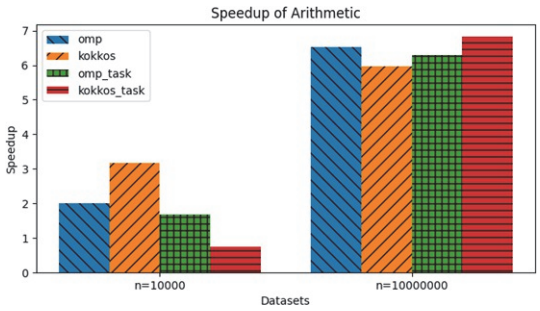

Fig. 7. Arithmetic application speedup.

E. *Prime*

Since this application belongs to the MapReduce class of problems and is also compute-bound, similar results can be expected. Fig. 8 shows the speedup graph for this application, which was tested using three datasets. In the graph, *l* denotes the beginning of the range, *h* the end of the range, and *s* is the multiplicative step, as explained in Section II. The differences between the OpenMP and Kokkos implementations are not significant and can be attributed to measurement errors.

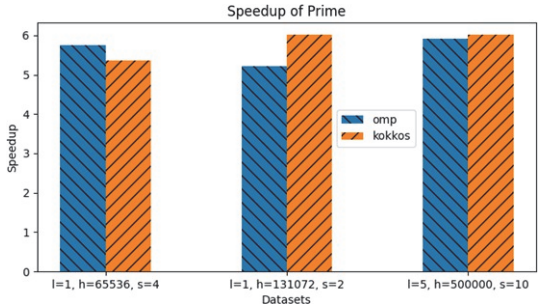

Fig. 8. Prime application speedup.

F. *Discussion*

The speedup achieved for the standard Kokkos and OpenMP implementations of the Arithmetic and Prime applications was certainly expected. These are the types of problems generally considered easy to parallelize [1]. However, the implementation using Kokkos tasks produced surprisingly good results. This programming model is primarily intended for solving problems where thread dependencies are non-trivial. In the case of MapReduce class problems, as described earlier, thread dependencies are reduced to a reduction operation at the end.

For the TPACF problem, the results were certainly not impressive. When every solution performs equally poorly, it is difficult to test and identify significant differences between the solutions. As shown in [7], solutions that use other parallel models executed on GPUs achieve much better speedups. Although this is beyond the scope of this work, it would be worthwhile to compare the Kokkos solution with these models. Of course, some modifications might be necessary in that case, but team-based parallelism is quite similar to parallel programming using the CUDA model. In that case, this solution should show better performance.

Furthermore, the differences in speedup between the OpenMP and Kokkos implementations for the SAD and SGEMM problems can be attributed to how the OpenMP *collapse* directive and the Kokkos *MDRangePolicy*

distribute work. In general, without additional specifications, the *collapse* directive combines nested loops to improve load balancing and cache reuse, while *MDRangePolicy* partitions the iteration space into tiles, introducing extra loop-bound computations and synchronization overhead. In SGEMM, this tiling, combined with lambda captures, can reduce compiler vectorization efficiency compared to the flattened OpenMP version. In SAD, OpenMP loops operate directly on contiguous memory, enabling better vectorization and cache locality. In contrast, Kokkos uses *Kokkos::View* abstractions that add index indirection and may misalign memory access patterns with cache lines. These factors together explain why, despite relying on OpenMP internally, Kokkos exhibits slightly lower CPU performance for fine- grained arithmetic and memory-bound kernels.

VI. Conclusion

Using a chosen collection of five benchmark applications, we assess the high-level Kokkos parallel programming model in this study. The performance of implementations created with the OpenMP programming model and those built with the Kokkos Core library was compared. Because the Kokkos Core library executes on CPUs through the OpenMP backend, significant performance gains were not observed. Nonetheless, the Kokkos Core library did well in several situations, particularly when it came to compute-bound and MapReduce-type tasks. In other cases, the discrepancies are attributed to the more complex usage of the Kokkos Core library and its distinct behavior compared to the OpenMP programming model.

There are several directions for future work. It would be useful to further optimize selected benchmark applications, especially using Kokkos *MDRangePolicy* and the Kokkos views. This can be achieved by tuning tile sizes to better fit CPU cache levels, adjusting memory layouts in *Kokkos::View* to improve data locality, and experimenting with *TeamPolicy* to expose additional parallelism within tiles. A comparative analysis with similar programming models and frameworks, such as RAJA and DPC++/SYCL, would also be beneficial on both CPUs and GPUs.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," 2006.

[2] Kokkos, "Documentation," https://kokkos.org/kokkos-core-wiki, accessed: 2024-10-02.

[3] N. Technology and L. N. Engineering Solutions of Sandia, "Machine model," https://kokkos.org/kokkos-core-wiki/ProgrammingGuide/ Machine-Model.html, accessed: 2024-10-02.

[4] 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.

[5] D. Patterson and J. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, ser. ISSN. Elsevier Science, 2017. [Online]. Available: https://books.google.rs/books?id= H7wxDQAAQBAJ

[6] 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.

[7] J. Đukić and M. Mišić, "An evaluation of directive-based parallelization on the gpu using a parboil benchmark," *Electronics*, vol. 12, no. 22, p. 4555, 2023.

[8] M. Mišić and M. Dodović, "An assessment of large language models for openmp-based code parallelization: a user perspective," *Journal of Big Data*, vol. 11, no. 1, p. 161, 2024.

[9] Kokkos, "Kokkos Ecosystem," https://kokkos.org/, accessed: 2024-10-02.

[10] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "Raja: Portable performance for large-scale scientific applications," in *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*. IEEE, 2019, pp. 71–81.

[11] R. Reyes and V. Lomüller, "Sycl: Single-source c++ accelerator program- ming," in *Parallel Computing: On the Road to Exascale*. IOS Press, 2016, pp. 673–682.

[12] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, no. 7.2, 2012.

[13] A. Minnaar, "Deep Learning GPU Performance Analysis: Memory Bound vs Math Bound Operations," https://alexminnaar.com/2020/04/11/dl-gpu-perf-memory-vs-math.html, accessed: 2024-10-02.

[14] Netlib, "BLAS (Basic Linear Algebra Subprograms)," https://www.netlib. org/blas/, accessed: 2024-10-02.

[15] W.-c. Feng, H. Lin, T. Scogland, and J. Zhang, "Opencl and the 13 dwarfs: A work in progress," in *Proceedings of the 3rd acm/spec international conference on performance engineering*, 2012, pp. 291–294.

[16] I. C. Laboratory, "CLOVER," https://icl.utk.edu/files/publications/2020/ icl-utk-1451-2020.pdf, accessed: 2024-10-02.

[17] A. Schuh, "Parboil Benchmarks," http://impact.crhc.illinois.edu/Parboil/ parboil.aspx, accessed: 2024-10-02.