PERFORMANCE AND PRODUCTIVITY EVALUATION OF KOKKOS

Karim Umar, Colin Arnet, Nicolas Winkler, Lasse Meinen, Saahiti Prayaga

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

This paper documents the evaluation of the performance portability framework Kokkos. The two benchmarking kernels LU-Decomposition and Deriche Edge Detection were chosen due to their varying levels of parallelizability and implemented using hybrid versions of Kokkos with OpenMP, CUDA, and MPI. Comparing the obtained performance on high-performance clusters and taking code complexity into account, we conclude that, although it is unable to keep up with the performance of other programming models by a constant factor, the simplicity of the Kokkos framework is a useful asset in the development of HPC applications.

1. INTRODUCTION

The multi-core revolution of computer architecture- in which multiple processors (cores) communicate directly through shared hardware caches- has introduced a tremendous opportunity to increase concurrency. To make effective use of this, it is vital to satisfy architecture-specific performance constraints for memory access patterns. However, taking a look at the current generation of machines in the High-Performance Computing (HPC) hardware landscape, one cannot help but notice the different programming models each of these machines are optimized to work for. Ranging from OpenMP 3, OpenMP 5, CUDA, HIP, DPC++, and OpenACC, the APIs these paradigms expose to the programmer are not uniform, and rewriting code for each of them to satisfy their respective memory constraints is a time-intensive process. This is where Kokkos steps in.

Kokkos, developed by Sandia National Laboratories, is a C++ programming model for performance portability across all major HPC platforms. Its implementation resides as a template library on top of programming models, providing abstractions for both parallel code execution and data management. As its website mentions, it is designed to target complex node architectures with N-level memory hierarchies, and can currently use CUDA, HPX, OpenMP and Pthreads as backend programming models. All in all, Kokkos' aim is to simplify programming for different parallel programming models. The programs only have to be written once and can then be compiled for different architectures by only changing compiler flags[1, 2, 3, 4].

In this paper, we document how we evaluated the performance of Kokkos by implementing two programs in Kokkos as well as other programming models and comparing their performance. The kernels implemented were chosen from PolyBench [5], a collection of benchmarks from various application domains of numerical computation. Our two kernels, LU-Decomposition and Deriche edge-detection, were chosen due to their varying levels of parallelizability and were implemented as MPI-hybrid versions of both Kokkos and OpenMP as well as in CUDA. These implementations were then tested on and optimized for a cluster of *Euler*, a central high-performance cluster of the ETH Zurich.

Related work. As Kokkos is a relatively new programming model, it remains to be determined whether its promise of simplicity is capable of truly matching the performance of other well-known programming models, such as OpenMP and MPI. Related work includes analysis of the performance of Kokkos against other performance portability frameworks, such as RAJA in the paper "*Evaluation of performance portability frameworks for the implementation of a particle-in-cell code*"[6].

Structure. The rest of this paper is structured as follows: Section 2 introduces notation and relevant concepts that will be used throughout the paper. In Section 3, we describe the implementations of the two kernels in their respective programming models. Section 4 then contains detailed experimental results and is followed by our conclusions in Section 5.

2. BACKGROUND

In this section, we introduce our chosen benchmarks and define the programming models we used to implement them.

2.1. Benchmark 1: LU-Decomposition

The LU decomposition is one of the basic tools of linear algebra. It decomposes a square matrix A into two triangular matrices L and U, such that $L \cdot U = A$.

The LU-Decomposition implementation from PolyBench makes use of the Gauss Elimination scheme to create the two triangular matrices row by row. This approach is not efficient for a parallel implementation because all workers would need to synchronize after every row. For this reason, we opted for a block-wise approach [7]. Many matrix operations benefit from processing data in blocks because it capitalizes on the spatial locality of accessing matrices row-by-row. Furthermore, if the blocks can be processed independently, this presents an obvious area to increase performance by distributing the blocks among multiple threads. Additionally, a row-by-row algorithm requires much more synchronization and communication between the threads and is therefore not suitable for parallelization. As in the PolyBench implementation, we implemented LU-Decomposition without pivoting.

Problem analysis. Similar to matrix multiplication[8], algorithms exist which are able to solve the problem of LU-Decomposition in less than $\mathcal{O}(n^3)$ (where the matrix has size $n \times n$) [9]. However, they are not used in practice. We stick to variations of the standard $\mathcal{O}(n^3)$ algorithm. Our implementations need $\mathcal{O}(n^3)$ floating point operations on a data size of $\mathcal{O}(n^2)$ ($(\frac{1}{6}n \cdot (1-3n+2n^2)$ additions and multiplications each and $n(n-1)$ divisions), which gives us an operational intensity $I = \mathcal{O}(n)$. For sufficiently large matrices, we can therefore assume an overall compute-bound operation[10] and should be able to benefit from parallelization.

Block algorithm. The main idea of the block-wise algorithm is the following: The matrices A, L, U are subdivided into four blocks.

$$\begin{pmatrix} L_{00} & L_{01} \\ L_{10} & L_{11} \end{pmatrix} \cdot \begin{pmatrix} U_{00} & U_{01} \\ U_{10} & U_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}$$

For a block size b, the submatrices have the following sizes:

$$\begin{array}{ll} L_{00}, U_{00}, A_{00} & b \times b \\ L_{01}, U_{01}, A_{01} & b \times (n-b) \\ L_{10}, U_{10}, A_{10} & (n-b) \times b \\ L_{11}, U_{11}, A_{11} & (n-b) \times (n-b) \end{array}$$

Given a matrix A partitioned as such, we can find the block-wise solutions for L and U as follows:

$$\begin{aligned} &\mathrm{L}_{00}, U_{00} := \mathrm{LU}(A_{00}) \\ &\mathrm{L}_{10} := \mathrm{TRSM}(U_{00}, A_{10}) \\ &\mathrm{U}_{01} := \mathrm{TRSM}(L_{00}^T, A_{01}^T)^T \\ &\mathrm{L}_{01} := \mathbf{0} \\ &\mathrm{U}_{10} := \mathbf{0} \\ &\mathrm{L}_{11}, U_{11} := \mathrm{LU}(A_{11} - L_{10} \cdot U_{01}) \end{aligned}$$

In the first step, we compute the LU-Decomposition of A_{00} sequentially. We get L_{10} and U_{01} with TRSM[1] operations with L_{00} and U_{00}. We then combine these results with a matrix multiplication and compute the rest of the decomposition recursively. The matrix multiplication forms the bottleneck of the algorithm. All the steps can be parallelized to improve performance.

All of our implementations use the described blocking principle.

[1] TRSM: triangular matrix solve operation

2.2. Benchmark 2: Deriche (Recursive Gaussian convolution)

The Deriche filter is a discrete approximation of a 2D infinite impulse response filter using a Gaussian function. This filtering algorithm is often used to generate the blurred input for canny edge detection[11].

In order to obtain the Deriche filter, the input image I is passed from left to right then from right to left according to the formulas:

$$\begin{aligned} y_{ij}^{+} &= a_1 I_{ij} + a_2 I_{ij-1} + b_1 y_{ij-1}^{+} + b_2 y_{ij-2}^{+} \\ y_{ij}^{-} &= a_3 I_{ij+1} + a_4 I_{ij+2} + b_1 y_{ij+1}^{-} + b_2 y_{ij+2}^{-} \end{aligned} \tag{1}$$

Where a_i and b_i are coefficients that depend on the standard deviation of the Gaussian function used, the derivation of these coefficients is detailed in [12], and y^+ and y^- are two temporary two-dimensional arrays for storing intermediate results, which are combined to obtain a new array:

$$\theta_{ij} = c_1(y_{ij}^{+} + y_{ij}^{-}) \tag{2}$$

In the second step of the algorithm, θ is used as the input and is passed from top to bottom according to the following formulas:

$$\begin{aligned} y_{ij}^{+} &= a_5 \theta_{ij} + a_6 \theta_{i-1j} + b_1 y_{i-1j}^{+} + b_2 y_{i-2j}^{+} \\ y_{ij}^{-} &= a_7 \theta_{i+1j} + a_8 \theta_{i+2j} + b_1 y_{i+1j}^{-} + b_2 y_{i+2j}^{-} \\ \theta_{ij} &= c_2(y_{ij}^{+} + y_{ij}^{-}) \end{aligned} \tag{3}$$

θ now contains the filtered input image.

Assuming the matrix size is $n \times n$, the asymptotic complexity of this algorithm is $\mathcal{O}(n^2)$, for each of the elements a constant number of operations is performed (13 Additions and 16 multiplications).

As one might suspect at first glance, the processed rows and columns are independent of each other, and the processing of these rows and columns can therefore be computed in parallel. Our sequential implementation is taken from the PolyBench implementation, and the same algorithmic concept is applied to our parallel versions of the Deriche Filter.

2.3. Programming models

In this section, we describe the different programming models that we used in our implementations.

MPI. Message Passing Interface (MPI) is an Industry standard for portable message-passing, designed to function on parallel computing architectures. MPI can be used for passing messages for parallel computation on distributed memory systems. There are several implementations of MPI (OpenMPI, MPICH, Intel MPI). By combining MPI with other programming models it is possible to extract more performance out of the nodes [13].

OpenMP. OpenMP is an API that provides shared-memory multiprocessing on a variety of architectures. With OpenMP, it is possible to parallelize programs within a shared-memory system. Because OpenMP works on shared-memory systems, it is not necessary to send messages like MPI. OpenMP starts threads that run the program concurrently which enables multithreading [14].

CUDA. Compute Unified Device Architecture (CUDA) is a parallel programming model that allows software to use NVIDIA GPUs. CUDA exposes an API to the programmer which lets them transfer data to GPUs and launch kernels that exploit the massive parallelism made possible by the GPU. CUDA is developed by NVIDIA and only works on NVIDIA GPUs [15].

Kokkos. Kokkos is a multiprocessing library developed to provide performance portability, the ability to write code and use the same code for multiple HPC architectures without making any performance sacrifices. Kokkos is implemented as a library and uses OpenMP and CUDA backends. Switching between compiling code for GPUs and CPUs only requires a compilation flag to be set if the code is written correctly. Kokkos also provides multidimensional arrays, called Views, that adapt to the corresponding hardware memory layout.

3. IMPLEMENTATION

In this section, we dive into the details of our implementations of the two benchmarks. The full source code is available on Gitlab[16].

3.1. Benchmark 1: LU-Decomposition

Baseline. Implementing the following algorithms, we started off with the PolyBench implementation as a baseline. The PolyBench implementation is simple; it is a simple triple loop in about ten lines of code. For the parallel implementations we make use of the block-wise approach mentioned in section 2.1. The matrix is divided into blocks:

$$\begin{array}{cc} \overbrace{\phantom{A_{00}}}^{b} & \overbrace{\phantom{A_{01}}}^{n-b} \\ A_{00} & A_{01} \\ A_{10} & A_{11} \\ \underbrace{\phantom{A_{00}}}_{n} & \underbrace{\phantom{A_{01}}}_{n} \end{array}$$

We chose the block size b to be a constant.

Matrix operations. Recalling the blocked algorithm, we first need to do the LU-Decomposition:

$L_{00}, U_{00} := \mathrm{LU}(A_{00})$

A quick back-of-the-envelope calculation shows that the size of the serial fraction of the code decreases with b. Therefore, we choose b to be sufficiently small. Following this, we perform two TRSM operations:

$L_{10} := \mathrm{TRSM}(U_{00}, A_{10});\ U_{01} := \mathrm{TRSM}(L_{00}^T, A_{01}^T)^T$

TRSM solves a triangular system of equations (L_{00}, U_{00}). It solves the system on multiple vectors (matrices: A_{10}, A_{01}). By working on each vector concurrently, the code can easily be parallelized. In the next step, we compute:

$A'_{11} := A_{11} - L_{10} \cdot U_{01}$

Out of all these steps, the most calculations need to be done in the matrix multiplication $L_{10} \cdot U_{01}$. We therefore tried carefully to parallelize this part well. These operations are parallelized in each programming model. In single-node implementations the calls to these operations are different from their respective multi-node implementations.

Single-node. For the single-node implementations, we apply the block algorithm as described above. The TRSM- and MM-operations can both be implemented as large, well-parallelizable loops.

We tested three different implementations of this algorithm.

OpenMP. MM and TRSM are parallelized using the `#pragma omp parallel for` directive.

Kokkos. The Kokkos implementation is more or less the same as the OpenMP variant; however, we used the `Kokkos::View` class to store the matrices and `Kokkos::parallel_for` to parallelize loops.

Kokkos DAG. For a second implementation using Kokkos, we opted to make use of `Kokkos::TaskScheduler`. This class allows us to submit C++ functors as tasks and define dependency relations between them. Tasks can split themselves up into smaller tasks. We make use of this feature and submit one top-level LU-task to the scheduler which, assuming the matrix is bigger than a certain threshold, will split itself up into one smaller LU-, two TRSM-, one MM- and a recursive LU-task. Each subtask will split into smaller pieces until a threshold on the problem-size is reached. By correctly marking the dependencies between all tasks, we can leave the parallel execution of all subtasks to the `Kokkos::TaskScheduler`.

CUDA. Similar to the OpenMP implementation, the CUDA version focuses on efficient implementations of TRSM and MM. For TRSM there is a thread for each vector that has to be solved. The MM is also done block-wise, where each entry of the result is handled by a thread. By using shared memory the memory performance is optimized.

MPI - Multi-node. Our parallelization strategy for MPI is to split the matrix into evenly sized blocks. These blocks

are then distributed among all ranks, that is, each rank owns a set of blocks. The rank owning the top left block will then start by performing LU-Decomposition sequentially on the block and broadcast the result. All ranks responsible for a block in either the top row or the left column will then perform the TRSM operation on the respective blocks. The resulting blocks will again be broadcast and used for the MM. Each rank will perform one $block \cdot block$ MM for each block it is responsible for. For the hybrid versions, the matrix operations are parallelized with the methods explained above.

3.2. Benchmark 2: Deriche

Baseline. The baseline used here is also the code implemented by PolyBench[5]: it consists of 2 sets of double-nested for-loops which compute according to the formulas given in the equation in section 2.2.

Kokkos. The implementation of the Kokkos-accelerated algorithm used parallelized for-loops to process the independent rows resp. columns concurrently.

OpenMP. The OpenMP-accelerated code was also implemented by parallelizing the loops within the calculation. The parallelization was done for single loops and nested loops using OpenMP directives.

MPI. Compared to LU decomposition, dividing the relevant data among the N nodes is straightforward. The computation on a $H \times W$ input matrix consists of 5 sections:

1. **Distribution**: The master node splits the input matrix into N sections, each section containing $\frac{H}{N}$ rows. A node i is sent the section i by the master node.
2. **filter 1** Each node computes the first step of the filtering algorithm, filtering from left-to-right and from right-to-left.
3. **Shuffle**: Each node splits its section vertically into N sections and sends the section j to node j.
4. **filter 2** Each node assembles a column of size $H * \frac{W}{N}$ from the $N-1$ other nodes and computes the second step of the filtering algorithm, filtering from top-to-bottom and filtering from bottom-to-top.
5. **Gather** Each node sends its column back to the master node, which assembles the data into the output image.

The code was further parallelized on a single node by using either Kokkos or OpenMP.

CUDA. The CUDA implementation follows the general form of implementing trivial parallel for-loops on a CPU. For each of the 4 filtering directions a CUDA kernel was written. The algorithm then spawns N threads, one for each column resp. row.

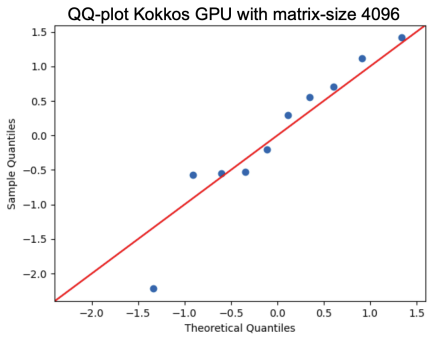


Fig. 1: QQ Plot for LU

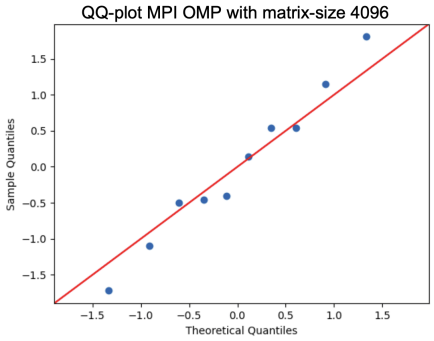


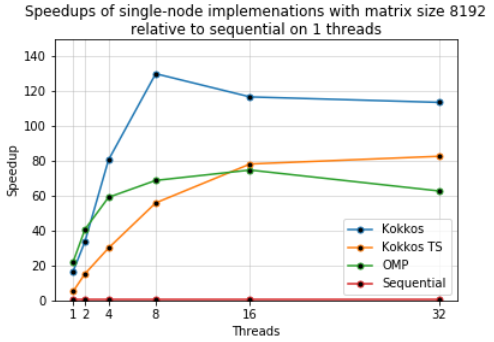
Fig. 2: QQ Plot for Deriche

4. EXPERIMENTAL RESULTS

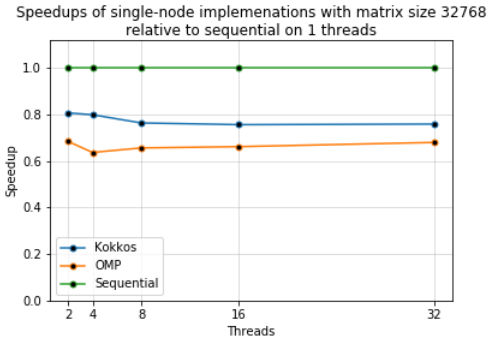
In the following paragraphs, we describe the experiment setup, both in terms of hard- and software, and present our experimental results.

Experimental setup. Each node of Euler VI has two 64-core AMD EPYC 7742 CPUs with 512 GB of DDR4 memory clocked at 3200 MHz. The system is interconnected by a dedicated 100 Gb/s InfiniBand HDR network [17]. GPU experiments were run locally on a GTX 1660super [18] complemented by an Intel I7-7700k with 16Gb of DDR4 memory clocked at 2133 MHz. All codes are compiled using GCC version 6.3.0 using the -O3 optimization flag. We used Open MPI 3.0.0 and Kokkos 3.3.1. The CUDA version used was 11.2. For Euler, allocated nodes were chosen by the batch system and no information regarding node topology was given.

Implementations were run on fixed input sizes. Multi-node, that is, MPI + OMP and MPI + Kokkos, implementations were run for varying numbers of ranks, with a fixed number of local threads per rank. Nodes were allocated only once for all measurements to ensure scaling behaviour would be representative. Single-node, that is, OMP, Kokkos and PolyBench, implementations were run for varying numbers of threads on a single node, allocated only once for all measurements. GPU implementations, that is, CUDA and

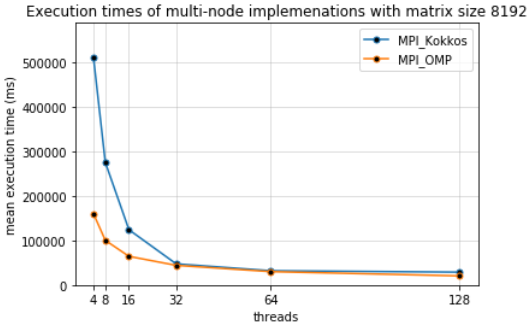


(a) LU

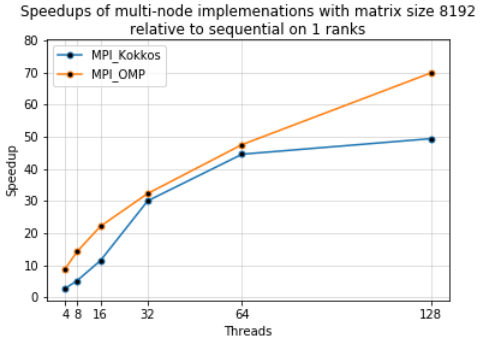


(b) Deriche

Fig. 3: Single-Node Implementation Performance Plots



(a) Time



(b) Speedup

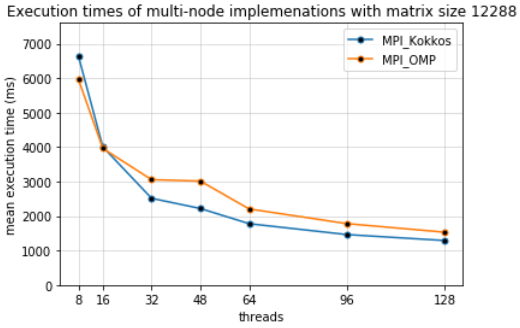
Fig. 4: LU Performance Plots

Kokkos, were run for varying problem sizes.

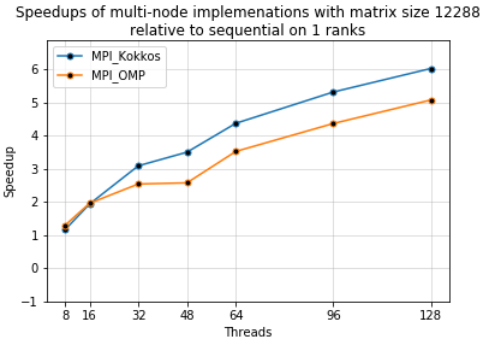
The execution time was measured using the ctime [19] library, part of the standard library. For measurements performed on GPUs, the measured time includes the transfer of data to and from the GPU's memory.

Results. Execution times were taken from the beginning until the end of the respective kernel functions. In particular, they do not include the generation of the random data we perform our operations on. We consider the mean execution time computed as the arithmetic mean with outliers removed using Tukey's method [20]. Experiments were run 20 times each and the 95% CI was within 5% of the mean. Measurement data was tested for normality using the QQ-method (for example, see Figures 1 and 2).

Fig.3 shows the respective speedups of the single-node implementations computed relative to the execution time of the serial implementation provided by PolyBench on one thread [5]. The lines in the LU plot represent the four implementations described in section 3.1. The line labelled "Kokkos" represents the Kokkos implementation that is structurally the same as the OMP implementation, that is, parallelized loops. "Kokkos TS" is the Kokkos implementation that makes use of `Kokkos::TaskScheduler`. We can see that the Kokkos analogue to the OMP implementation scales significantly better with more cores. We assume this is because of better defaults for parallel loops (e.g.

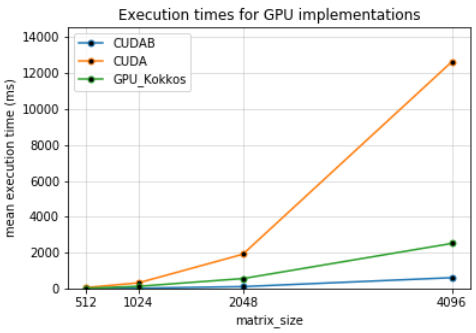


(a) Time

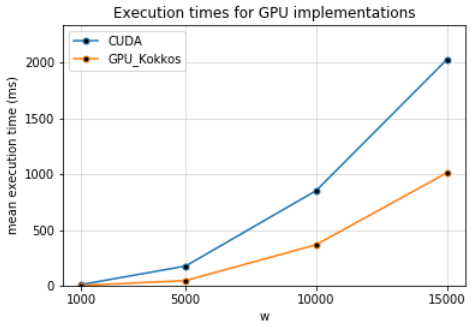


(b) Speedup

Fig. 5: Deriche Performance Plots



(a) GPU LU



(b) GPU Deriche

Fig. 6: GPU Performance Plots

chunk size, thread allocation, dynamic/static scheduling). It is interesting to see that both implementations of LU using Kokkos have an improved speedup as thread-count increases compared to the one using OMP. As for the Deriche (see Figure 3b), we were disappointed to see that the single-node implementations did not scale even at very large matrix size, this is almost certainly due to the low operational intensity of the filter operation, meaning the operation on a single node is memory-bound. Therefore, parallelization even slows down the computation due to parallel overhead.

Fig.4 and Fig.5 show the execution times and respective speedups of the multi-node implementations computed relative to the execution time of the serial implementation provided by PolyBench on one thread [5]. We can see that for both the LU and the Deriche kernel, MPI + OMP scales similarly to MPI + Kokkos. We observe MPI+OMP outperforms Kokkos by a more or less constant factor. We assume this is because of more memory movements caused by Kokkos' abstractions. The speedup for both kernels is, as expected, sublinear. This confirms the statements made by Kokkos' developers that Kokkos does not outperform OMP. [21]

Fig.6 shows the respective execution times of the GPU implementations. For the LU kernel, the execution times for the naive CUDA implementation and the optimized, "blocked" implementation are shown. We can see that for both the Deriche and LU kernels. The compiled Kokkos code outperforms a naive CUDA implementation by a significant margin. For the LU-Decomposition however, optimizing the CUDA code resulted in an implementation that was even faster than the Kokkos implementation. We could not improve upon our CUDA implementation of Deriche, the NVCC performance profiler showed that although the kernel executions of the bare CUDA implementation outperformed the Kokkos kernels, the Kokkos implementation spent significantly less time transferring data between the host and the graphics card, a feat we were not able to emulate.

5. CONCLUSIONS

The goal of Kokkos is to simplify the development of HPC applications by only writing one implementation for different programming models. With the LU-Decomposition and Deriche Edge Detection, we implemented two kernels with Kokkos and different programming models. In both kernels, Kokkos showed good scaling behavior for more threads. Compared to the other programming models, we often observed that in many cases Kokkos couldn't keep up performance-wise by a slight constant factor. For the single-node LU-Decomposition, Kokkos achieved better performance than OpenMP. The reason for the performance difference is that Kokkos is unable to make use of programming-model-specific optimizations. Although Kokkos does not achieve the same performance as the native programming models, it provides a simple way to write code for multiple architectures at once.

6. REFERENCES

[1] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," 2013.

[2] kokkos git repo. Accessed: 14.10.2021. [Online]. Available: https://github.com/kokkos/kokkos

[3] kokkos kernels git repo. Accessed: 14.10.2021. [Online]. Available: https://github.com/kokkos/kokkos-kernels

[4] kokkos tutorial slides. Accessed: 14.10.2021. [Online]. Available: https://www.osti.gov/biblio/1525572

[5] Polybench. Accessed: 14.10.2021. [Online]. Available: http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[6] V. Artigues, K. Kormann, M. Rampp, and K. Reuter, "Evaluation of performance portability frameworks for the implementation of a particle-in-cell code,"

Concurrency and Computation: Practice and Experience, vol. 32, no. 11, p. e5640, 2020.

[7] V. S. Rana, M. Lin, and B. Chapman, "A scalable task parallelism approach for lu decomposition with multicore cpus," 2016.

[8] V. Strassen, "Gaussian elimination is not optimal," *Numerische mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[9] J. E. H. James R. Bunch, "Triangular factorization and inversion by fast matrix multiplication," *Math. Comp.*, vol. 28, pp. 231–236, 1974.

[10] M. Püschel. (2021, Oktober) Lecture 3b: Design of parallel and high performance computing, slide 6. Accessed: 07.10.2021. [Online]. Available: https://moodle-app2.let.ethz.ch/mod/resource/view.php?id=654799

[11] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, 1986.

[12] R. Deriche, "Using canny's criteria to derive a recursively implemented optimal edge detector," *International journal of computer vision*, vol. 1, no. 2, pp. 167–187, 1987.

[13] Open mpi. Accessed: 10.10.2021. [Online]. Available: https://www.open-mpi.org/

[14] Open mpi. Accessed: 10.10.2021. [Online]. Available: https://www.openmp.org/

[15] Cuda. Accessed: 10.10.2021. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[16] K. Umar, C. Arnet, N. Winkler, L. M, and S. Prayaga. Source code. [Online]. Available: https://gitlab.ethz.ch/umarka/dphpc_project

[17] Polybench. Accessed: 06.10.2021. [Online]. Available: https://scicomp.ethz.ch/wiki/Euler

[18] Gtx-1660-super specs. Accessed: 06.10.2021. [Online]. Available: https://www.nvidia.com/en-us/geforce/graphics-cards/gtx-1660-super/

[19] Std chrono documentation. [Online]. Available: https://en.cppreference.com/w/cpp/chrono/high_resolution_clock

[20] J. W. Tukey *et al.*, *Exploratory data analysis*. Reading, Mass., 1977, vol. 2.

[21] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.