Kokkos: Portable Performance

Alex Hocks
alex.hocks@tum.de
Technical University of Munich
Germany

ABSTRACT

Considering modern developments, where computer architecture and its performance are changing quickly, the performance and portability of software projects becomes more important. In this paper we will present Kokkos, a portability framework, that provides the means to achieve this goal, show which efforts are necessary to port to Kokkos and what its capabilities are.

1 INTRODUCTION

In light of recent developments of rapid performance gains on different hardware, achieving the highest HPC performance requires switching to different platforms. When the underlying hardware changes drastically, such as from CPU to GPU, then porting is inevitable. But porting large scale projects with over a million lines of code to another system requires tremendous amounts of work. Another downside is that for each project multiple code bases exist, with basically the same content. Modifying the project will require modifying all code bases while finding the corresponding parts. This is not ideal. To mitigate this, we will present Kokkos, a portability library with its own ecosystem, and its performance. By using Kokkos further porting efforts should be minimal and the code bases should be able to merged into a single one.

2 KOKKOS: A PORTABILITY FRAMEWORK

Kokkos was originally developed by Sandia National Laboratories to abstract applications from the underlying hardware architecture, especially in the HPC environment. Meanwhile, it is more than only a programming library. It has a big ecosystem, that not only contains the core programming model, but also extensions for math and graph support. Further parts are interoperability layers for Python and Fortran as well as tools for debugging, profiling and tuning. [15]

Kokkos Kernels is the ambiguous name of the extension for math and graph support and should not be confused with the kernels of the core programming model. In many engineering and physics related problems linear algebra is necessary. Therefore, vendors create specialized libraries for their own platform, resulting in poor portability. Kokkos Kernels provides generic implementations, that can delegate to vendor implementations. It also contains uncommon functionality, which encourages vendors to create specialized implementations. [15]

HPC applications often use more than one language such as Python, Fortran and C++. A common approach is that Python coordinates the application and delegates computationally intense tasks to parts written in C++. With Fortran being one of the oldest programming languages, many HPC applications are written with it. To encompass new functionality, those also delegate tasks to parts written in C++. To ease this process, Kokkos provides interoperability layer for both: Fortran Language Compatibility Layer and pyKokkosbase. [15]

For better analysis Kokkos offers a unified tool system. It introduces negligible overhead and the use of tools does not need recompilation. With these tools users can see profiling information about timing and memory independently of the underlying hardware. Due to combined efforts of vendors and the Kokkos developers, internal hardware related information is viewable in a user readable way. Because of that, big applications using Kokkos are also profilable without further efforts. Beyond profiling the tool system offers automatic optimization. This can change parameters within the Kokkos runtime system, but also user defined parameters. [15]

Another aspect of Kokkos is its wide variety of user support. It offers lectures including exercises, a Wiki and an active communication platform for Kokkos related questions. [15] For gaining a deeper understanding we made use of all of those and find them all to be helpful.

Kokkos' main purpose is portability. As Kokkos Core, the programming model, is a C++ library, it provides templated abstractions to enable reusing the same code on different platforms. On a higher level these abstractions are either execution or memory related. These are: Execution Spaces, Execution Patterns, Execution Policies, Memory Spaces, Memory Layouts and Memory Traits. [16]

2.1 Execution Basics

Execution Spaces represent an environment in which parallel operations will be handled in. Such spaces are `Kokkos::Cuda` or `Kokkos::Thread`. These stand for a Cuda stream or thread pool respectively. Every Execution Space has a FIFO queue for parallel operations. Each parallel operation can specify in which Execution space it should run in. By doing so, scheduling performance can be increased. For all Execution Space types Kokkos creates default instances. These are used as a fallback solution in case the user does not specify one for a parallel operation. In this case there are no guarantees that different parallel operations are executed at the same time. [16]

Execution Policies define in which manner parallel operations are executed. Since a parallel operation is a set of "work items" [16] a policy defines the range in which work should be done. Additionally, it contains information about which Execution Space to use. To address the parallel nature of said operations, Execution Policies can also define scheduling strategies. [16] A basic policy is *RangePolicy*. It can be used to specify a range of n items to be worked on. [5]

Execution Patterns is the Kokkos term for sections of code executed in parallel, also called kernels, and are the priorly mentioned parallel operations. These take shape as `parallel_for`,

parallel_reduce, parallel_scan. The body of Execution Patterns must be independent from execution order, since no such guarantees are made. In general, kernels can be used by calling the wished type. The parameters that all have in common are an Execution Policy and a construct with code to be run in parallel. This construct can either be a lambda or a functor. In the following we will call this lambda. When using a *RangePolicy* in DefaultExecutionSpace for the ExecutionPolicy it can be replaced by an integer. Parallel_for represents a regular for loop, in which all loop iterations are executed in parallel. Each loop iteration is only executed a single time. The lambda argument is the current index. [16] An example using CUDA execution space can be seen in listing 1.

```
// Kernel with indices in [0, n)
parallel_for(RangePolicy<Cuda>(0, n),
  KOKKOS_LAMBDA(int i) { /* ... */});
```

Listing 1: Short Kokkos kernel using RangePolicy in CUDA execution space. Based on [16]

Parallel_reduce functions just as parallel_for only with the addition that the output of all loop iterations is aggregated into a single output. The lambda has as parameters the current index and as many references to accumulators as there are Reducers. The Execution Patterns function has as parameters additionally all needed Reducers. A Reducer contains information about reduction result initialization and combination as well as a reference to the output destination. Kokkos includes Sum, Prod, Min, Max, MinMax, MinLoc, MaxLoc, MinMaxLoc, LAnd, LOr, BAnd and BOr as Reducers. When using Sum, it can be replaced by a reference to the output destination, since Sum is the default Reducer. The user can also define Reducers. [16] Listing 2 shows a simple reduction kernel with an explicit Sum reducer storing the output in result. It calculates the sum over the first n values multiplied with two.

```
double result = 0.0;
parallel_reduce(RangePolicy<OpenMP>(n),
  KOKKOS_LAMBDA(int i, double& s)
        { s += 2 * i; }, Sum{result});
```

Listing 2: Kokkos kernel using RangePolicy in OpenMP execution space and summing over the first n values multiplied with two. Based on [16]

Parallel_scan maps inputs a_i to the aggregation of $a_0 \dots a_i$ with an associative operator. Due to parallel execution each iteration can be executed multiple times. [4, 16] The lambda parameters are an index, an accumulator, and a Boolean variable, that specifies whether the current pass is final. The Boolean variable is necessary to only overwrite data when no other pass needs it anymore. [16] Listing 3 shows a kernel, that maps value a_i to $2 \cdot (a_0 + \dots + a_i)$.

2.2 Memory Basics

Memory Spaces are a domain from which users can request memory. They are an abstraction from the underlying Hardware, therefore also from memory allocation and management. Different spaces

```
parallel_scan(n,
  KOKKOS_LAMBDA(int i, int64_t& partial_sum,
        bool final)
        {
          partial_sum += 2 * array(i);
          if (final) array(i) = partial_sum;
        });
```

Listing 3: Kokkos kernel using implicit RangePolicy in default execution space and performing a scan operation. Based on [16]

can only be accessed from certain hardware. *CudaSpace*, *CudaUVMSpace* and *CudaHostPinnedSpace* can be accessed from GPUs, whereas *CudaUMV-Space* and *HostSpace* from CPUs. To transfer data from one space to another or to create copies, Memory Spaces provide the deep_copy function. [16]

Memory Layouts define in which order multidimensional array indices are used to calculate the actual memory address. Since Kokkos provides abstractions for Memory Layouts, algorithms can be written independently of them to accommodate the needs of different hardware. [16]

Memory Traits define additional behavior when accessing memory. Example Traits are atomic access and unmanaged, with the latter removing overhead imposed by the Kokkos runtime system while reducing portability. [16]

Views bring the prior three aspects together as being the most common data structure in Kokkos. A View is an array or pointer of up to rank N. Dimension sizes can be specified during compile and runtime. To create a View the data type must be specified as pointer or array. By using a pointer or array the dimension size will be determined at run or compiletime respectively. When a View is created, all necessary memory will be allocated by the run time. During definition of a View the Memory Space, Memory Layout and Memory Trait can be specified to add further details. Since a View is a pointer to allocated memory, it handles reference counting and automatic deallocation just like std::shared_ptr. Therefore, a View does not need to have its own allocated memory but can point to another Views memory. [16] Listing 4 contains examples of Views with run and compiletime dimensions as well as explicit Memory Spaces and Memory Traits.

```
// simple 2D View in default memory space
// with runtime dimensions N and M
auto v2d = View<double**>{"v2d", N, M};
// same thing but with compile time dimensions
auto v2dc = View<double[5][5]>{"v2dc"};
// 3D View in CudaUVMSpace and both dim types
auto v3d =
  View<int**[5], CudaUVMSpace>{"v3d", N, M};
// Atomic access to v3d
auto v3da = View<int**[5], CudaUVMSpace,
  MemoryTraits<Atomic>>{v3d};
```

Listing 4: Examples of Kokkos View instantiations. Based on [16]

2.3 Advanced Techniques

Atomic access on memory can also be guaranteed regardless of data size. Based on hardware and data type Kokkos uses atomic operations or compare-and-swap or a shared lock table. The later hashes the requested address into the lock table. [16]

In the example of particle simulations, where each entity excerpts a force on each other, the parallel calculation of this can lead to race conditions. A common approach for mitigation in CPU algorithms is data replication. On GPUs with their limited memory, this is not viable. Therefore, atomic operations are usually used on accelerators. This approach also runs on CPUs, however GPUs have special function units that fulfill this purpose resulting in higher performance. Another data structure is `ScatterView`. It is an abstraction for the presented approaches allowing synchronized write access to memory that is potentially used by other threads. During parallel access a participant can contribute data. `ScatterView` handles this request by using one of the two options. After completion of the kernel data can be merged. With atomic operations it is already merged. [16]

Often Views are not just one dimensional. An example is the initialization of a three-dimensional View. Assuming its size is 100 in each dimension, parallelizing the process along a single dimension results in poor performance on GPUs because its large parallel capacities are not facilitated. Using a simple Kokkos range to index the entire one million entries requires the use of fixed operations to deconstruct the flattened index into its three components for the different dimensions. This approach fixes performance issues on GPUs, but has a fixed function from a flattened to the three dimensional space. The performance of this approach depends on the underlying hardware. On a CPU the additional overhead is suboptimal. `MDRangePolicy` addresses this issue and is capable of covering multidimensional view spaces. Additionally, it provides functionality to split the address space into tiles, which allows for more abstract access patterns. [16]

When loops are not tightly nested, the parallelization with `MDRangePolicy` will fail, due to dependencies between threads. In cases where groups of threads share certain resources, *Basic Thread Teams* can be used. With `TeamPolicy`, an Execution Policy, teams can be created. Each team has a pool of threads. In this case each loop iteration is not directly assigned to a thread, but rather to a team. This approach is able to run nested loops concurrently. The parameters of `TeamPolicy` define the amount of teams and their respective thread pool size. The lambda parameters are extended by a reference to the active team. For Kokkos to recognize the nested loops, the inner loops execution policy is `TeamThreadRange`. [16]

Upon creation of a parallel operation with a `TeamPolicy`, the threads of all teams are active. If there are multiple nested loops in sequential order, synchronization might be necessary because a faster thread can already proceed to the next nested loop while another thread is still working on the first loop. To synchronize all threads and let them wait until all are at the same point in execution, a *barrier* can be used. If necessary, each team can request *scratch memory*, that is accessible from each thread within the team. Furthermore, every thread can also request memory for itself. Due to backend limitations, requests must be stated prior to team creation. When possible, *scratch memory* is accelerated by existing hardware.

```
parallel_for(TeamPolicy<>(N, AUTO),
 KOKKOS_LAMBDA(const team_t& team_h)
 {
  int i = team_h.league_rank();
        parallel_for(
        TeamThreadRange(team_h, K),[&](int j)
        {
         parallel_for(
         TeamVectorRange(team_h, L),
         [&](int j){ /* do something */});
         team_h.barrier();
         parallel_for(
         TeamVectorRange(team_h, M),
         [&](int j){ /* do more */});
         });
 });
```

Listing 5: Hierarchical Kokkos kernel using vectorization and barrier for synchronization. Based on [16]

Further parallelization is achievable with vectorization. Kokkos supports one form of implicit vectorization with the Execution Policy, `ThreadVectorRange`. This can be used within a `TeamThreadRange` policy. However, this method only gives hints to the compiler for vectorization. To enforce vectorization, the use of explicit SIMD types is necessary. SIMD data types are handled differently depending on the platform. On CPUs computation and storage operates on the same vector data type. GPUs store data as arrays, while each computational unit only uses a single scalar variable. Therefore, Kokkos offers a templated wrapper for SIMD types. [16] Listing 5 displays a possible way to create a hierarchy of kernels. To access the current team index, `league_rank` is called. In the middle *barrier* is called to synchronize all threads progress.

On a higher level parallelism is expressed through Execution Spaces. Upon inserting kernels into distinct Execution Space instances, kernels are potentially executed concurrently. Using a *fence* on a Execution Space ensures that all enqueued kernels are completed. [16]

Due to driver overhead, on-device scheduling and memory bandwidth limitations, kernel execution time is increased by a certain latency. If higher performance is necessary, but no further kernel and memory optimizations are possible, then Kokkos Graphs can amortize some latency. Kernels, that are often used in conjunction with each other or need to be computed sequentially, can be placed in a Graph. When executing this the mentioned latencies still exist for the Graph. But in comparison to executing the kernels individually, latencies are reduced overall. [16]

3 PORTABILITY PERFORMANCE

The main interest of this part lies in two things. Firstly, there already exist many large scale HPC applications such as Uintah [14]. To fully facilitate the new features and better performance of new computer architectures, projects should be ported to those systems. However, the resulting effort to accomplish this can be tremendous. Therefore, the question arises: What are the steps to port to Kokkos? Secondly, after porting to Kokkos, how well does a Kokkos

application perform in comparison to other or more specific implementations and how portable is code using Kokkos?

3.1 Porting Effort

As the first step, code that can be run in parallel has to be identified and refactored into Kokkos Kernels. As the C++ STL and other data structures are not portable, they must be replaced with data structures provided by Kokkos. Initially, unmanaged Kokkos `Views`, i.e. `Views` with the unmanaged memory trait, can be used to wrap the existing data structures. By doing so, the refactoring effort can be minimized, which allows for gradual porting to Kokkos. However, unmanaged `Views` are not portable. Therefore, after initial tests they must be replaced with other alternatives. [14]

3.2 Performance analysis using case studies

In the following we will look at different case studies about Kokkos regarding performance, portability, and other factors. This should give insight in the capabilities of Kokkos, but also what its drawbacks can be.

3.2.1 Uintah. To solve gas and fluid dynamics problems, complex mathematical computations are necessary. Uintah is a software that finds solutions for such problems. It provides a runtime system to solve user applications, while separating runtime and application code bases. Initially, application developers had to write different code for CPU, Xeon Phi and GPU tasks. The usage of Kokkos enabled higher parallelism and the merge of three different code bases into one. [14]

Since Uintah is a large project, converting it to mainly use Kokkos kernels was a significant amount of work. Most of it consisted of the mentioned steps: finding loops, converting them into kernels and transforming data structures into Kokkos usable ones. Sunderland et al. [14] advise to perform refactoring gradually. The unmanaged memory trait supports achieving this. By wrapping existing data structures in unmanaged Kokkos `Views`, other runtime or application code does not need to undergo major refactoring and it is easier to find non Kokkos API compliant code. To verify kernel portability, they can be extracted into a test environment with mock inputs. [14]

The Arches application for Uintah was converted to using Kokkos with fairly little effort allowing fast first tests with Kokkos. Sunderland et al. [14] mention two exemplary kernels: Upwind and van Leer. In comparison to the prior standard implementation both achieved positive speedup, whereas Upwinds speedup is greater due to less computational branches in the kernel. The performance improvements are a result of Kokkos' `parallel_for` and memory access pattern as well as rewriting kernels. [14] The detailed results are in table 1.

Patch size	8^3	16^3	32^3	64^3	128^3
Upwind Kokkos Speedup	4.6	10.0	10.7	12.9	12.7
van Leer Kokkos Speedup	2.76	4.05	4.04	5.01	6.37

Table 1: Table of speedups of Upwind and van Leer in comparison to their serial implementation. Source: [14]

In another more detailed example within the Arches application the transformation and performance of a kernel for heat dissipation calculations is displayed. [8, 14] The standard implementation contains a for loop over Uintah arrays using iterators. These arrays are indexed using `IntVector`, which represent a tuple of three elements. Due to implementation details of `IntVector`, pointer indirection occur. This provides ease of development with the downside of worse performance. [14] The initial version can be seen in listing 6. Said arrays are D, X, Y, Z, phi and rhs. The first five are input data and rhs is the output destination. The variables of type `IV` extract the indices and the section with the assignment to rhs performs the actual calculation.

```
typedef IntVector IV;
for(Iterator itr(low, high); !itr.done(); ++itr)
{
  IV c = *itr;
  IV xp = c+IV(1,0,0), xm = c+IV(-1,0,0);
  IV yp = c+IV(0,1,0), ym = c+IV(0,-1,0);
  IV zp = c+IV(0,0,1), zm = c+IV(0,0,-1);
  rhs[c]+=
   ax * (X[xp]*(D[xp]+D[c]) * (phi[xp]-phi[c])
    -X[c] * (D[c]+D[xm]) * (phi[c]-phi[xm]))
  +ay * (Y[yp]*(D[yp]+D[c]) * (phi[yp]-phi[c])
    -Y[c] * (D[c]+D[ym]) * (phi[c]-phi[ym]))
  +az * (Z[zp]*(D[zp]+D[c]) * (phi[zp]-phi[c])
    -Z[c] * (D[c]+D[zm]) * (phi[c]-phi[zm]));
}
```

Listing 6: Initial sequential verions. Source: [14]

Basic mitigation is done by using Kokkos `parallel_for`, rewriting the indexing mechanism and using unmanaged `Views` as wrappers. [14] This is the first kernel in listing 7. Based on the parameter list, we infer that range is a `MDRangePolicy`. The loop structure was replaced with `parallel_for` and the index extraction with `IV` was replaced with range. The calculation is fundamentally unchanged. Further improvements are gained by rewriting the basic Kokkos kernel into using auto vectorization. For this, subviews need to be generated, such that a nested `parallel_for`, which allows auto vectorization, can operate independently of external indices. [14] It appears in the second half of listing 7. Here the `MDRangePolicy` was altered, such that the range of the third dimension was sliced into parts for vectorization. Additionally, subviews are created to support vectorization.

With Kokkos kernels high speedups are achievable in comparison to the basic Uintah implementation, whilst having portable code. By using implicit vectorization an additional speedup of around factor two can be reached. Considering the portability of Kokkos code, speedups of up to 50x are achievable. [14] These results are displayed in table 2. Similar speedups using SIMD are also stated in [12].

The presented approach improves performance on a single system. Holmen et al. [1] examine the effects of using MPI and Kokkos to parallelize across multiple nodes. As scalability, following Amdahls law, depends on the amount of work, through the use of

```
// basic kokkos kernel
parallel_for(range ,[=](int i, int j, int k){
  rhs(i,j,k) +=
  ax * (X(i+1,j,k)
  * (D(i+1,j,k) + D(i,j,k))
  * (phi(i+1,j,k) - phi(i,j,k))
  - X(i,j,k)
  * (D(i,j,k) + D(i-1,j,k))
  * (phi(i,j,k) - phi(i-1,j,k)))
  + ay * (Y(i,j+1,k)
  * (D(i,j+1,k)+D(i,j,k))
  * (phi(i,j+1,k) - phi(i,j,k))
  - Y(i,j,k)
  * (D(i,j,k) + D(i,j-1,k))
  * (phi(i,j,k) - phi(i,j-1,k)))
  + az * (Z(i,j,k+1)
  * (D(i,j,k+1) + D(i,j,k))
  * (phi(i,j,k+1) - phi(i,j,k))
  - Z(i,j,k)
  * (D(i,j,k) + D(i,j,k-1))
  * (phi(i,j,k) - phi(i,j,k-1)));
});

// SIMD kokkos kernel
parallel_for(range, [=](int i, int j,
pair<int,int> k_range){
  auto r = subview(rhs, i, j, ALL());
  /* generate other subviews */
  parallel_for(krange, [&](int k){
    r(k)+= ax*(xp(k)*(dp0(k)+d00(k))
                *(pp0(k)-p00(k))
    -x0(k)*(d00(k)+dm0(k))
                *(p00(k)-pm0(k)))
    +ay*(yp(k)*(d0p(k)+d00(k))
                *(p0p(k)-p00(k))
    -y0(k)*(d00(k)+d0m(k))
                *(p00(k)-p0m(k)))
    +az*(z(k+1)*(d00(k+1)+d00(k))
                *(p00(k+1)-p00(k))
    -z(k)*(d00(k)+d00(k-1))
                *(p00(k)-p00(k-1)));
  });
});
```

Listing 7: Conversion to (un-)vectorized Kokkos kernels. Source: [13]

Kokkos high scalability is achievable without the need to increase work. [1]

3.2.2 Comparison Kokkos vs. others. Alternatives for Kokkos for HPC programming are OpenMP, OpenACC, CUDA and RAJA. OpenMP focuses on multicore CPU programming while using annotation style directives. OpenACC shares similarities with OpenMP in its style, but its main target is GPUs. Using CPUs to parallelize

		32^3		64^3		128^3	
		ms	x	ms	x	ms	x
Serial Uintah		1.06	1.0	8.04	1.0	64.9	1.0
Kokkos Serial	Naive	0.65	1.6	4.30	1.9	36.1	1.8
	SIMD	0.31	3.4	2.47	3.3	20.2	3.2
Kokkos 4 Threads	Naive	0.17	6.4	1.16	6.9	8.94	7.3
	SIMD	0.08	13	0.58	14	5.27	12
Kokkos 16 Threads	Naive	0.07	16	0.54	15	4.51	14
	SIMD	0.04	24	0.31	26	2.54	25
Kokkos 32 Threads	Naive	0.04	29	0.28	29	3.52	18
	SIMD	0.02	43	0.16	49	3.42	19
Kokkos CUDA	GPU	0.09	12	0.21	38	0.61	105
	SIMD	0.09	12	0.21	38	0.63	103

Table 2: Times and speedups of Kokkos kernels with and without vectorization. Source: [14]

is possible, but might need refactoring to achieve comparable performance. [3] CUDA only supports the usage of GPUs. [11] RAJA is another portability library just as Kokkos. [3]

In Artigues et al.s [3] portability performance analysis of the different libraries a plasma physics problem was used as the testing scenario. Their baseline implementation was initially written in FORTRAN but was ported to C++. The baseline implementation was parallelized into different versions. These were made with either OpenMP, OpenACC, CUDA, Kokkos, a combination of OpenMP and Kokkos, or RAJA. Each version was created with only the use of the respective library. The exception is the implementation with OpenMP and Kokkos. It uses Kokkos' unmanaged `Views` for memory management and OpenMP for parallel tasks. These versions were analyzed regarding performance, but also in other aspects such as portability, productivity, and readability. [3]

Since OpenMP and OpenACC use annotations as compiler directives for parallelization, we will not describe these libraries in further detail than in listing 8. [10] [2] It displays two short examples of the usage of annotations in OpenACC and OpenMP.

```
// simple parallelized loop
// using OpenACC
#pragma acc parallel loop
for(unsigned int i=0; i < nCount; i++)
  // do something in each thread

// simple parallelized loop
// using OpenMP
#pragma omp parallel for
for (int i = 1; i < n; i++)
  c[i] = a[i] + b[i];
```

Listing 8: Examples for annotations of OpenACC and OpenMP. Based on [10] [2]

Due to their use of annotations, code written with these two libraries has high readability. The programming model of Kokkos and RAJA are quite similar. Both support the three mentioned

parallel operations. However, Kokkos provides functionality for reductions of vectors with `ScatterView`. RAJA does not. This type of reduction was necessary for solving the testing scenario. [3]

A benefit of Kokkos is, that it provides defaults for most of its templates, which in turn results in portability without the need of preprocessor commands, unlike RAJA. In Artigues et al.s [3] Kokkos implementation the only exception for that, was the memory access pattern for `ScatterView`. In RAJA no defaults are provided, forcing the programmer to think about what to choose. Furthermore, that creates many branches in source code, what reduces readability. Readability is further reduced in RAJA, because RAJA only accepts lambdas as methods to provide parallel code. In Kokkos the user can also define functors. This helps isolating code and improving reusability. Nevertheless, in comparison to OpenMP or OpenACC both Kokkos and RAJA require more work to use, since code must be placed in lambdas or functors. Using CUDA needs a similar amount of work. In terms of library specific additional code, the order from least to most is: OpenMP and OpenACC, Kokkos and CUDA, RAJA. Productivity wise, the time to grasp the concepts of Kokkos and RAJA is similar. The process of developing OpenACC code is like CUDA to equivalent execution and data mappings. Regarding portability, with Kokkos and RAJA the implementations, that are designed to run on CPUs, do not need modifications to run on GPUs. When using Kokkos, it allows compilation with a Makefile with architecture dependent flags for optimization. With the other libraries, this needs to be done by hand. Due to their use of templates, Kokkos and RAJA generate code optimized for a specific platform at compile time. Therefore, debugging, or simply inspecting code must be done on assembly level. [3] Based on these findings, an overview of the non performance aspects of different libraries can be seen in table 3.

criterion	OpenMP	OpenACC	CUDA	Kokkos	RAJA
code clarity	high	high	low	medium	medium
productivity	high	medium	low	medium	medium
portability	low	medium	low	high	high
performance	high	high	high	high	medium

Table 3: Evaluation of libraries in selected categories. Source: [3]

Looking at the performance results of the CPU benchmarks, OpenACC is the fastest, closely followed by OpenMP. Kokkos is about 1.21 times slower than OpenACC. A potential reason is the overhead of Kokkos' runtime system. Out of all RAJA has the worst performance. The version, in which OpenMP and Kokkos were used, has a similar performance as the native OpenMP version. This shows that if more performance is wished, Kokkos kernels can be partially replaced with alternatives of other libraries. [3]

GPU performance is slightly different. OpenACC and CUDA are in the first two places. CUDA implementations benefit from newer hardware. On average the RAJA implementation is faster than the Kokkos version. [3]

The results for Kokkos in Artigues et al.s [3] analysis are that, it provides a high level of abstraction and a variety of predefined functionality. A downside is that compilation results can only be viewed in assembly. But the code can be compiled for CPUs or GPUs without the need of modifications. The provided default values however are useful. Furthermore, almost no preprocessor commands are needed. Another benefit, that increases productivity, is that Kokkos is well documented. Performance results are within acceptable range in comparison to the platform specific libraries and on CPU Kokkos scales well with core count. [3]

3.2.3 High Energy Physics (HEP). Tests at the LHC can be classified under high energy physics. Each HEP experiment there produces large data sets. This necessitates high computational power to run simulations using the data. Only using CPUs might not be sufficient for this purpose. Therefore Dong et al. [6] ported a selected simulation to use GPUs. They created two versions. One uses CUDA and the other one Kokkos. The CUDA version was created by mainly parallelizing over a loop that handles particle calculations. STL data structures were converted into arrays. Data transfers from and to the GPU were minimized. Due to intermediate testing results, they created a from the original CUDA code derived version that groups calculations for certain particles together. By doing so, performance is expected to improve. We will not go in further detail. For more information see [6]. With these two versions the speedup ranges between a factor of 1.2 and 8 depending on input data. [6]

When CUDA is used as the parallel device in Kokkos, then code from both libraries can interact with each other without limitations. This allows Kokkos kernels to use CUDA functions and access memory, that is managed by CUDA. Because of that CUDA code can be gradually converted to using Kokkos. The Kokkos version made use of this option. Dong et al. [6] chose to do so to explore portability as well as the problems, that can arise when porting a potentially large project from serial C++ to CUDA and then to Kokkos. Each porting procedure from one library to another was done with as few changes as possible. During the conversion from CUDA to Kokkos, the result of each rewrite was validated. The first step of this conversion was to replace CUDA kernels with Kokkos kernels. Memory allocated with CUDA was not altered. The change to using `Views` was done as the next step. [6]

The performance of the Kokkos version was measured with the CUDA version as its baseline. Tests were run with different backends. [6] The results can be seen in table 4. Clean initializes a large array. Sim_a is a bigger kernel with the main logic. Sim_ct is a smaller reduction kernel. Copy d->h is the time for data copy operations from devices to host. The names until now are parts of the Event loop. It in turn is the previously mentioned parallelized loop. [6]

Looking at the results, the Kokkos version performs in the majority of all cases worse than the native CUDA implementation. When comparing the Kokkos version with the CUDA backend against its baseline, its performance is on average 56% worse. However, in two of the cases it is only about 17% and in other two about 35% worse. With Clean being the exception, this performance is much closer to the baseline. Comparing the CUDA backend version against the initial serial CPU version, it still delivers a significant speedup of more than factor 4. For the CPU backends the copy d->h test shows high speed since the device equals the host. The serial backend is overall

	Kokkos (relative)				CUDA
Kernel	CUDA	Serial	pThread	OpenMP	μs
Clean	2.83	7.67	1.61	2.26	14.0
sim_a	1.17	36.9	11.2	11.5	49.2
sim_ct	1.35	12.9	1.66	1.99	15.2
copy d->h	1.30	0.02	0.06	0.04	25.3
event loop	1.16	7.26	2.61	2.79	321
event loop speedup vs. CPU	4.63	0.74	2.06	1.93	5.38

Table 4: Speedup of respective Kokkos version (left) in comparison to native CUDA implementation (right). Source: [6]

worse than the native serial implementation by 26%[1]. Both parallel CPU backends perform roughly the same, while being worse than the native CUDA version but about twice as fast as the native serial version. Dong et al. [6] conclude that initializing device memory with Kokkos has some overhead, because first host `Views` are created before this is copied to the device. But despite the overhead from `Views` and from launching kernels, overall, it is close enough to the CUDA version. Furthermore, they state that the portability Kokkos offers, is more significant than minor performance losses. [6]

3.2.4 Deep Neural Networks (DNN). Common use cases for DNNs are recognition and classification tasks in AI. [7] For brevity we will omit the underlying concept. Introductory information can be found in a paper by Maind et al. [9]. Neural Networks can be modeled by using matrices. Therefore, Ellis et al. [7] conducted experiments, that implement image recognition using the Kokkos Kernels library. Since the matrices had dimensions of at least 1024 by 1024 up to 65536 by 65536 and the number of matrices was between 120 to 1920, sparse matrices were used. Based on this, the performance of three versions were compared. One was a serial reference implementation. Two parallel versions were developed using the respective libraries Kokkos, or LAGraph. The Kokkos version (KKV) was written in a way, such that it is portable, which was not mandatory for the tests. The Kokkos and LAGraph versions perform 300 to 500 times better than the reference version. KKV is faster than the LAGraph version (LAGV) when the matrix dimension is 4096 by 4096. For tests with the largest matrices it is the other way around. KKV scales linearly with matrix size, until the indexing data size needs to be changed. Attempts of scaling up KKV onto multiple nodes, result in higher performance until 12 nodes. In their implementation work is statically distributed to the different nodes. Due to the nature of sparse matrix multiplication, after a certain amount of calculations the work of each node differs. Because nodes need to synchronize, a faster node must wait for a slower one leading to bad scalability regarding node count. Therefore, Ellis et al. [7] propose to split work dynamically to maintain an even amount of work across all nodes. [7] Finally, they conclude that using Kokkos Kernel for matrix multiplication increases performance, but "[...] scaling to higher nodes counts shows diminishing returns." [7].

[1]Correction: In [6] 36% was stated, but considering the values in table 4 26% would be consistent

4 SUMMARY

In the beginning, we described what Kokkos is and how to use it. Beyond that, the porting effort to use Kokkos is comprised of the three steps: locating code, refactoring to kernels and replacing data structures. The main take away points of the case studies are the following: In Uintah the porting steps were displayed in an example. Compared to serial code, high performance is achievable with rather little effort. With the use of vectorization speedups of up to 50x are possible. For Uintah scalability across multiple nodes was done with no complications. In the comparison of Kokkos against other alternatives, it was shown that Kokkos' readability is lower. However, it provides many default and predefined components, which enables high portability. Performance is also high, but it may vary. Depending on the backend another alternative may be faster. In HEP we focused on the CUDA backend of Kokkos. The perforance is worse than a native CUDA implementation, but still provides significant speedups compared to serial code. Additionally, it is pointed out again that its high portability outweighs minor performance losses. In DNN the performance of the Kokkos Kernels library for linear algebra was evaluated. Very high performance gains are achievable. In this case, horizontal scalability was low, due to the nature of sparse matrices.

5 CONCLUSION AND FUTURE WORK

We have shown the concepts and vast functionality of Kokkos. Understanding enough to parallelize own code might take time. The described case studies show that performance of Kokkos kernels is mostly faster than serial C++ code. In comparison to other parallel implementations its performance can be better or worse depending on the current work load. Benefits of using Kokkos are its supplied data structures and access patterns, allowing sequential porting and high portability. With the use of SIMD data types and similar methods, vectorization is possible with only minor changes. Furthermore, Kokkos is capable of integrating well with other systems. Scaling to higher node counts varies from case to case. Special work distribution may be necessary. In comparison to other frameworks readability is more impacted and the porting effort is higher, but its acceptable performance and high portability make Kokkos a compelling option. Since ARM based architectures are becoming more performant, further studies can show if the HPC landscape can easily transition by using Kokkos and whether performance is comparable.

REFERENCES

[1] 2017. Improving Uintah's Scalability Through the Use of Portable Kokkos-Based Data Parallel Tasks. https://doi.org/10.1145/3093338.3093388

[2] Sara Royuela Alcazar. 2018. High-level Compiler Analysis for OpenMP. (4 2018), 181.

[3] Victor Artigues, Katharina Kormann, Markus Rampp, and Klaus Reuter. 2019. Evaluation of performance portability frameworks for the implementation of a particle-in-cell code. 32 (5 2019). https://doi.org/10.1002/cpe.5640

[4] Guy E Blelloch. [n. d.]. Prefix Sums and Their Applications. ([n. d.]).

[5] Irina Demeshko, Jerry Watkins, Irina K Tezaur, Oksana Guba, William F Spotz, Andrew G Salinger, Roger P Pawlowski, and Michael A Heroux. 2019. Toward performance portability of the Albany finite element analysis code using the Kokkos library. 33 (3 2019). https://doi.org/10.1177/1094342017749957

[6] Zhihua Dong, Heather Gray, Charles Leggett, Meifeng Lin, Vincent R. Pascuzzi, and Kwangmin Yu. 2021. Porting HEP Parameterized Calorimeter Simulation Code to GPUs. 4 (2021). https://www.frontiersin.org/article/10.3389/fdata.2021.665783

[7] J. Austin Ellis and Sivasankaran Rajamanickam. 2019. Scalable Inference for Sparse Deep Neural Networks using Kokkos Kernels. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)* (2019-09). https://doi.org/10.1109/HPEC.2019.8916378

[8] Risi Imre Kondor and John Lafferty. [n. d.]. Diffusion Kernels on Graphs and Other Discrete Input Spaces. ([n. d.]).

[9] Sonali B Maind and Priyanka Wankar. 2014. Research Paper on Basic of Artificial Neural Network. 2 (1 2014).

[10] Farber Rob. 2022. *Chapter 1: From serial to parallel programming using OpenACC*. https://learning.oreilly.com/library/view/parallel-programming-with/9780124104594/B9780124103979000019.xhtml

[11] Farber Rob. 2022. *First Programs and How to Think in CUDA*. https://learning.oreilly.com/library/view/cuda-application-design/9780123884268/B978012388426800001X.xhtml

[12] Damodar Sahasrabudhe, Eric T. Phipps, Sivasankaran Rajamanickam, and Martin Berzins. 2020. A Portable SIMD Primitive Using Kokkos for Heterogeneous Architectures. In *Accelerator Programming Using Directives* (2020), Sandra Wienke and Sridutt Bhalachandra (Eds.). Springer International Publishing. https://doi.org/10.1007/978-3-030-49943-3_7

[13] Daniel Sunderland, Brad Peterson, and John Schmidt. 2016. Performance Portability in the Uintah Runtime System Through the Use of Kokkos. (11 2016).

[14] Daniel Sunderland, Brad Peterson, John Schmidt, Alan Humphrey, Jeremy Thornock, and Martin Berzins. 2016. An Overview of Performance Portability in the Uintah Runtime System through the Use of Kokkos. In *2016 Second International Workshop on Extreme Scale Programming Models and Middlewar (ESPM2)* (2016-11). https://doi.org/10.1109/ESPM2.2016.012

[15] Christian Trott, Luc Berger-Vergiat, David Poliakoff, Sivasankaran Rajamanickam, Damien Lebrun-Grandie, Jonathan Madsen, Nader Al Awar, Milos Gligoric, Galen Shipman, and Geoff Womeldorff. 2021. The Kokkos EcoSystem: Comprehensive Performance Portability for High Performance Computing. 23 (9 2021). https://doi.org/10.1109/MCSE.2021.3098509

[16] Christian R. Trott, Damien Lebrun-Grandie, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. 33 (4 2022). https://doi.org/10.1109/TPDS.2021.3097283