Performance Portability for HPC Applications through the RAJA abstraction layer

Philipp Pindl
philipp.pindl@tum.de
Technical University of Munich
Munich, Germany

May 2022

Abstract This paper gives a brief overview of the core concepts behind the RAJA abstraction layer and how it can be used to write performance portable code. Multiple examples, where RAJA was employed as a means of making HPC applications portable are considered along with performance measurements. These results are used to evaluate RAJAs ability to make code portable. Additionally, we then discuss the ease of portability, as well as in how far RAJA impacts the performance of ported applications.

1 Introduction

In recent times various accelerators, like GPUs, have become increasingly common in today's HPC architectures, like in the currently leading supercomputer FRONTIER [1]. Each of FRONTIER's nodes has 4 AMD GPUs in addition to a CPU for tasks like machine learning [2]. However current HPC architectures use multiple GPUs from different vendors. The majority of used GPUs are manufactured by NVIDIA, AMD, and Intel. However each of these GPUs has multiple frameworks and tools for programming them. This faces developers with the problem of writing code that remains portable across multiple architectures, that ideally maintains its performance even when ported. Code for HPC architectures is usually complex and can contain on the scale of 10^6 lines of code, like ALE3D [3], a physics simulation tool. Having to rewrite an entire codebase of this size each time hardware changes take place, will be time consuming in addition to downtimes during which the code is unable to run. This slows down research process and in turn delays the publishing of results. In order to address this problem several performance portability frameworks, such as Kokkos and RAJA, have emerged in the last few years, which promiss to achieve portability across GPU and CPU architectures. In this paper we will consider the RAJA framework in detail. RAJA was developed by the Lawrence Livermore National Laboratory further referred to as LLNL and used to prepare various of their applications for the arrival of El Capitan [4] which is projected to be the fastest supercomputer at the time of its installation in 2023. These ported applications include the ARES Multiphysics code, ALE3D, and the neutral particle transport code ARDRA.

2 Overview of RAJA

RAJA is an abstraction layer for C++ code that comes in the form of multiple software libraries. It has had its first GitHub release in 2016 and has been actively growing on the site since [5]. RAJA seeks to make a single-source code performance portable across many heterogenous HPC architectures, through parallelizing loops on different platforms. To achieve this multiple backends are used to compile the C++ program for a corresponding ar-

chitecture. Most notably RAJA offers support for CUDA, HIP and SYCL which are used for running code on NVIDIA, AMD and Intel GPUs respectively. Furthermore, parallel CPU models are supported via the SIMD and OpenMP backends. This resolves the need for programmers to maintain multiple versions of their source code each using CUDA or HIP directly and targeting a specific GPU or CPU. Especially for large HPC applications which contain on the scale of 10^6 lines of code this involves a lot of superfluous work. [6]

To accomplish this RAJA offers building blocks to abstract standard C++ loops. This approach leaves the loop body itself unchanged thus keeping the necessary source code changes to a minimum. More specifically a loop can be expressed in terms of an Execution Policy, an Iteration Space, and a Traversal Template. We now elaborate on each of these concepts further using an example. This simple C++ loop just adds a scalar multiple of the vector b to the vector a:

```
for (int i = 0; i < N; ++i)
{
    a[i] += c * b[i];
}
```

The equivalent RAJA version looks like this:

```
using namespace RAJA;
RangeSegment seg (0, N);
forall<loop_exec> (seg, [=] (int i) {
    a[i] += c * b[i];
});
```

The first thing to note is that the loop body stays the same the only difference being the conversion to a C++ lambda, which takes the loop index as a parameter. We call this lambda the lambda kernel body. A kernel in the context of GPUs is the loop body of a traditional loop. In the following the different components of the RAJA implementation are explained:

1. Execution Policy: In our case the `loop_exec` execution policy is used. An execution policy most importantly describes which execution backend and execution platform to use along with information for compile time checking. Execution policies can be defined by the users themselves. However, the RAJA library already defines many commonly used Execution Policies. In our case `loop_exec` describes a sequential policy for the CPU that does not force any optimization like SIMD but tries to incorporate them.

2. Iteration Space: Iteration spaces define the range of the loop as a set of loop indices. Here `seg` is the iteration space used for the loop containing the indices from 0 to $N - 1$.

3. Traversal Templates: Traversal templates provide a description for how the lambda is applied for each index in the iteration space. In our case `forall` resembles a classic for loop where the lambda is applied once for every index. However, RAJA also provides reduction and scan operations. [7]

It is also worth mentioning that RAJA offers abstractions for memory called views which provide different layout options and multi-dimensional indexing. Some of these layout options can be used for enhanced performance by automatically permuting data. Along with views there are two complementary libraries for RAJA, namely Umpire and CHAI, which also deal with memory management. Umpire offers a unified approach for (de)allocating, moving, and copying memory, as well as memory pools and strategies for allocation thus providing high memory performance [8]. CHAI on the other hand comes with a managed array type that is responsible for automatically copying memory between memory spaces, like the CPU and GPU. [6]

3 Performance and Portability asessment

Now that the core principle of the workings of RAJA have been established, we will take a look at some examples where RAJA has been used as an abstraction layer to achieve portability. Specifically, we will use the RAJA Performance Suite [9], SW4 and the implementation of a particle-in-cell code (PIC) with

the RAJA abstraction layer. For each of these we will first describe the methodology that were used and the measurements that were taken. At the end of each section we will then discuss the difficulties that accompanied porting the code and the resulting implementation in terms of portability, if such information is present in each case. To conclude, we analyze the performance deviation of the RAJA implementations from non portable approaches or the base implementation depending on what data is available for each study.

3.1 RAJA Performance Suite

The RAJA Performance Suite contains a collection of kernels that can be used to assess the performance of a RAJA implementation and compare it to reference implementations, which are for example written in CUDA or OpenMP. In a study from 2019 the LLNL used this suite to compare a Sequential, OpenMP and CUDA implementations to their corresponding RAJA generated counterparts. All variants were executed on several HPC architectures which contain between 20 and 98304 nodes. It should be noted that data allocation/deallocation, initialization and necessary transfers are not included in the runtime measurements. Figure 1 shows the performance difference for each variant against its reference implementation as a histogram. [7]

A spike at the 0% difference mark is apparent for all three variants, thus indicating that for most kernels the RAJA and reference implementations perform about equally well. OpenMP displays a notable increase in variance compared to both other variants. The LLNL study concluded this might be due to the C++ compilers having difficulties optimizing OpenMP pragmas together with C++ template abstractions which RAJA uses. Overall, in 55% of all cases the performance of the RAJA implementation was found to be within 10% of the reference implementation's performance. The paper does not mention any difficulties in the process of rewriting the applications in RAJA. Therefore, we can draw no conclusions on the ease with which portability can be achieved. [7]

3.2 PIC code

In a study from 2019 the "Max-Planck-Institut für Plasmaphysik" together with the "Max Planck Computing and Data Facility" used the Kokkos [10] and RAJA frameworks to develop an HPC application for solving a problem from the plasma physics domain. This problem consists of solving the Vlasov-Maxwell [11] system of equations using a Particle-In-Cell method [12]. The Kokkos framework is a similar framework to RAJA and strives to make applications performance portable as well, by using the same concepts as RAJA. The performance of both implementations was then compared against each other and against OpenMP, OpenACC [13] and a OpenMP/Kokkos hybrid variant. The performance of the different implementations was measured on a multicore CPU socket and a single GPU. To run on a multi core CPU RAJA was compiled with the OpenMP backend. The results are shown in Figure 2 and 3 respectively.

In this paper Artigues et al. report that a two-step process was necessary for refactoring: Firstly, arrays were replaced with RAJA views. Secondly, the loop bodies were moved into C++ lambdas and the for loop was adjusted by using the `forall` traversal template and iteration spaces. It is noted that RAJA forces the user to explicitly state which platform it is run on in the code. This leads to the necessity of two branches for the CPU and GPU version. Also, RAJA does not automatically allocate memory for views which makes it the responsibility of the user to do so. The RAJA implementation of the PIC code was obtained by rewriting a baseline C++ version of the code. Artigues et al. report that "significant restructuring" was required due to the necessity of lambda functions. Furthermore, before running the performance tests, it was discovered that the RAJA implementation accesses the memory in such a way that leads to false sharing of cached data. This occurs when two threads are working on data stored in the same cache line. If one of the threads writes data to the cache the whole cache line and thus the data of the other thread is invalidated. This means that the second thread must synchronize the whole cache line despite the fact none of its data has been

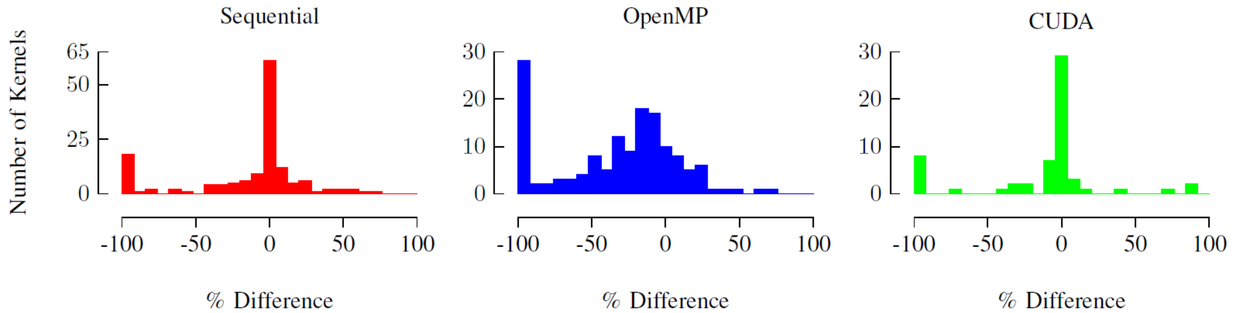

Figure 1: Performance difference (%) between RAJA and reference variants of Performance Suite kernels on five HPC platforms. Positive values mean that the RAJA variant is faster than the reference. From [7]

changed. The RAJA implementation did not allocate the memory as a block in contrast to the other implementations, resulting in this phenomenon. This can be seen as a drawback of RAJA as it is the user's responsibility to address this issue as a result of the manual memory allocation in RAJA.

For the tests on the GPUs shown in figure 3 compute time and data transfer time was measured separately since depending on the application data transfers might only happen in large intervals. Since the GPUs used were all manufactured by NVIDIA the CUDA backend for RAJA was used for compilation. In this test scenario RAJA performance is much better than on the CPU. On all GPUs RAJA's compute time is about a factor of between 2 and 2.5 slower than the fastest implementation in each case. Furthermore, on all GPUs its performance is about even to Kokkos performance. [14]

3.3 SW4

SW4 [16] is a seismic wave propagation model developed to carry out 3-D seismic modeling. In a study [15] on performance portability from 2021 this was one of the applications whose performance portability was evaluated. The performance of the RAJA implementation was measured across three GPUs from different vendors. Two high performing GPUs from AMD and NVIDIA and one weaker integrated Intel GPU. It is worth mentioning that, although this paper deals with applications for HPC architectures the results for the Intel GPU are still worth considering, at least for the portability aspects of RAJA. This is important as Intel seemingly intends to get into the HPC market, since the announcement of their Ponte Vecchio GPU. In the coming years Intel GPUs could become more common in the leading HPC architectures as a result of this. However the performance results on Intel HPC GPUs may be different than those measured on the integrated GPU. [17]

In order to compare the performance portability, roofline efficiency was used as a way of quantifying the ratio of achieved and theoretically possible performance. Roofline efficiency is calculated by dividing the measured FLOPS by the theoretically achievable performance as derived from the roofline model. The roofline model evaluates the maximum performance in terms of two possible limiting factors: Either a computation is memory bound or compute bound. In order to run on each of the three GPUs the CUDA, HIP and SYCL backends were used to obtain versions capable of running on each GPU. The study was limited to a single important kernel used in SW4's computations.

Table 2 shows the results from the measurements. Most notably the column FR_k denotes the average Flop Rate of the kernel and P_k stands for the peak Flop Rate measured. It is noticeable that the version run on the Intel GPU does display a much smaller Arithmetic Intensity AI_k, which describes how much FLOPS can be carried out per data transfer. The pa-

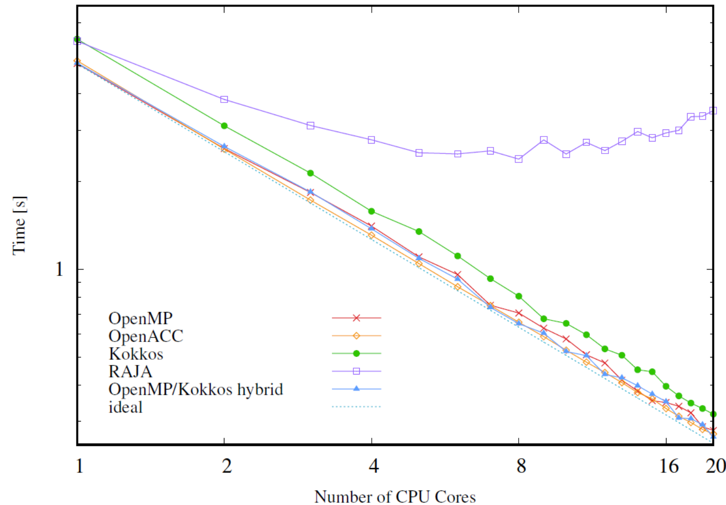

Figure 2: Log-log plot of the compute times per iteration as functions of the number of CPU cores. We compare the performance of Kokkos, RAJA, and OpenACC on the CPU to plain OpenMP. The codes were run on a 20-core Intel Xeon Gold 6148 2.4GHz (Skylake) CPU. The times shown were averaged over 10 runs. From [14]

GPU	RAJA Backend	SW4 commit hash	Kernel Runtime [s]
AMD MI100	HIP	ff9225f	20.4
NVIDIA A100	CUDA	ff9225f	6.77
Intel Gen9	SYCL	f2f45db	3620

Table 1: SW4 version and kernel runtime. From [15]

GPU	AI_K	FR_k (GF/s)	Bound	P_k (GF/s)
Intel Gen9	0.101	4.86	Memory	7.11
NVIDIA A100	3.73	1620	Memory	4700
AMD M100 (est)	0.744	536.	Memory	692

Table 2: SW4 roofline-based performance data. From [15]

per attributes this to the fact that the RAJA SYCL backend is still in development and therefore not optimized for register spilling. It is also mentioned that the CUDA backend is the most tuned out of the backends, matching with the highest measured Arithmetic Intensity of all three variants. [15]

Table 3 shows the roofline efficiencies of RAJA (SW4) on each platform in comparison to other applications ported with the help of different frameworks like Kokkos or OpenMP. It should be noted that this does not serve as a good direct comparison, as every framework was used to port a different set of kernels. [15]

This paper also does not point out any complaints with rewriting the kernel in RAJA and successfully manages to port the application to three different GPU architectures. [15]

3.4 Portability Analysis

In this section RAJA will be evaluated in terms of its portability and ease of use. In all of the discussed ex-

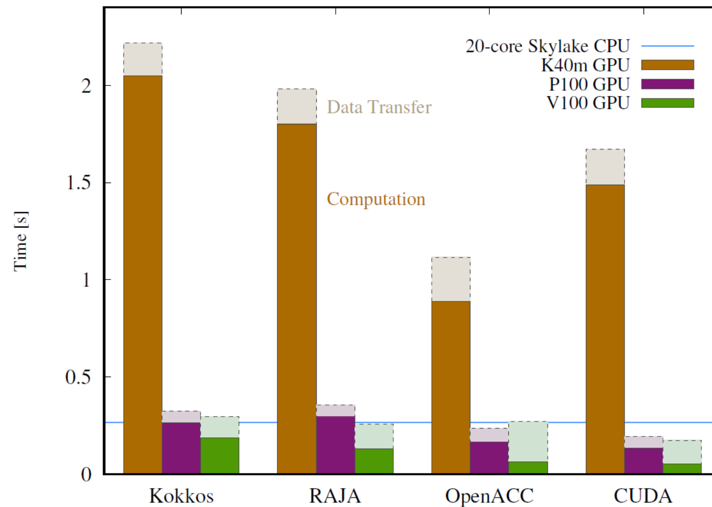

Figure 3: Plot of the times per iteration for various GPU models. The performance of Kokkos, RAJA, and OpenACC to CUDA was compared. In addition to the compute time the time necessary for one data transfer is shown. The codes were run on a NVIDIA K40m, a NVIDIA P100, and a NVIDIA V100 GPU. The times shown for the GPUs were averaged over 200 runs. From [15]

amples the code successfully compiled and executed for each targeted platform. The tested platforms include various GPU architectures from NVIDIA, AMD and Intel and a multi-core CPU architecture. This shows that RAJA is truly portable across many architectures through its various backends.

The ease of portability was only discussed by the PIC-code example in section 3.2. The study finds that "significant restructuring" was required due to the necessity of lambda functions [14]. Furthermore because of memory management being the responsibility of the user, mistakes like false sharing mitigation in this case can happen easily and more effort is required for porting an application. Apart from this RAJA provides a clear way of refactoring loops with its library as discussed in section 2.

3.5 Performance Analysis

Now the the performance of ported RAJA applications will be evaluated based on the measurements from the previous sections.

First up the results from section 3.1 will be discussed. Here it is indicated by Figure 1 that the performance of the majority of CUDA and sequential kernels are performance portable, whereas the performance for the OpenMP variants is shown to be more unreliable. This implies that the OpenMP backend is more susceptible to performance anomalies, which should be considered when it is used. However, since these tests were performed in 2019, this high variance is likely to have decreased because both C++ compilers and RAJA have undergone active development and therefore have been improved since then.

In section 3.2 the measurements in figure 3 show that for GPUs RAJA accomplishes its goal of performance portability to a certain degree. This can be seen from the acceptable performance overhead in comparison to the native implementations, of about a factor of two. However for HPC applications, with a runtime of several hours, a factor of two might still be significant enough to justify an implementation in one of the faster frameworks, although this needs to be decided on a per-case basis. On the other

Application	Kernel	Intel Gen9	Nvidia A100	AMD MI100	Perf.Port Metric(%)	Std.Dev. /Avg	Min /Max
AMR-Wind	MLABec	74.2	79.1	42.8	60.6	0.302	0.541
	MLPoisson	34.9	54.1	8.47	18.2	0.705	0.157
	MLNode	37.7	34.1	34.4	35.3	0.057	0.904
HACC	BarExtras	69.7	45.5	83.5	62.1	0.291	0.544
	Corrections	94.6	36.5	89.1	60.9	0.437	0.385
	DuDt	71.3	49.1	75.8	63.0	0.218	0.648
	Geometry	80.9	55.3	74.1	68.3	0.189	0.683
SW4	curvelinear4sg	68.3	34.5	77.5	53.0	0.377	0.445
RI-MP2	RIMP2$omp	28.4	18.6	4.50	9.64	0.700	0.158
XSBench	XSBench$omp	61.2	32.7	23.1	33.2	0.508	0.377
TestSNAP	FusedDeiDrj	65.1	35.5	9.99	20.9	0.748	0.153
	Ui	38.8	21.7	2.36	6.06	0.871	0.061
	Yi	5.11	27.0	10.5	9.15	0.804	0.189

Table 3: Roofline efficiency, E_k (%) and Performance Portability Metrics. From [15]

hand however, the tests on the CPU, shown in figure 2, indicate RAJA lacking behind significantly as more CPU cores are used. This stands in contrast to the other implementations including Kokkos, that scale similarly to the ideal curve. RAJA's inability to scale accordingly can be attributed to the false sharing, discussed in section 3.2, despite having been improved in the executed version of the code. Artigues et al. remark that further improving the performance of RAJA would require a great amount of tuning. [14] However this stands in direct contrast to the goals of portability as only having one code version was meant to reduce the amount of necessary work. This suggests, that for RAJA to make an application truly performance portable across all architectures, it would be necessary to put a lot of effort into the porting process.

The last case study in section 3.3 makes it evident, based on Tables 1 and 2, that RAJA was capable of achieving acceptable performance on all architectures, as the measured FLOPS do not deviate from their theoretical peak by a factor of more than three. It should be mentioned that this theoretical peak does not correspond with the FLOP rate of a reference implementation, that is expected to also have a lower value than the peak. Table 3 displays these deviations in greater detail. It is shown that the NVIDIA GPU has the lowest efficiency of 34.5% despite having the shortest execution time of 6.77 seconds. This may imply that the NVIDIA hardware was utilized the worst out of the three GPU types. The performance portability metric is calculated as the harmonic mean of the ratio between measured and peak FLOPS for each of the three GPUs. For the SW4 kernel RAJA achieves a value of 53.0%. This value may however vary greatly for different Kernels, as can be seen at the example of the AMR-Wind application in the first row. For this app the AMReX framework [18] achieves 60.6% and 18.2% for different kernels respectively. Because of this the performance portability metric for RAJA might not be reflected accurately by the obtained result and further research must be done to receive a more solid value. However, should the performance portability metric remain around the 50% mark for different kernels, RAJA does indeed provide performance portability for all three GPU types. Further insightful information can be obtained from this table by

comparing the SW4 against the TestSNAP row. TestSNAP was ported using the above mentioned Kokkos framework, whose similarities to RAJA make it an interesting contender. Under the assumption that the measured performance portability metric for RAJA is accurate, RAJA does outperform Kokkos for every kernel, in the sense that it utilizes the given hardware resources better. Kokkos highest measured performance portability metric is only at about 21% which shows RAJA to be the significantly faster framework. Overall, RAJA's performance is in the middle of all results making it a viable solution to the problem of portability.

All in all there is no deciding factor for when a framework is truly performance portable, which leads to one having to make per-case decisions whether RAJA is performance portable enough for the application. With that said the results from 3.2 clearly show that RAJA did not scale nearly as well as other frameworks when executed on a multi-core CPU with the OpenMP backend. This was attributed to memory management errors. As can be seen in all examples RAJA does perform much better when executed on a GPU architecture. The results of the RAJA performance suite show little performance difference between reference and RAJA implementations for most kernels. For the PIC-code a performance overhead of about afactor of 2 worse than the reference implementation was found. This suggests that RAJA is a good option for performance portability in the case of GPU architectures. For CPU architectures it should be used with care, as it is possible to performance to greatly suffer.

4 Conclusion

We conclude that RAJA is a mature performance portability framework. More specifically RAJA offers an approach to performance portability that does not require changing the loop bodies in most cases as described in section 2. Furthermore, RAJA does achieve portability on a great variety of CPU and GPU platforms as evident from the presented examples. As can be expected from a platform independent framework, the performance of RAJA implementations does not reach that of native implementations. RAJA did not scale well on an CPU architecture and therefore should not be relied on for performance critical computations on CPUs which a great number of HPC programs are. The performance overhead of about a factor of two for GPU architectures, measured in the PIC code, displays a far better result, but sill amounts to a significant difference for applications with runtime that lasts over a few hours. However as observed in the other two examples the performance difference is not always this great. Nevertheless a performance overhead of this size cannot simply be neglected and developers have to weigh up the importance of faster execution time against portability. Especially when maintaining large programs with over 10^6 lines of code that would take multiple persons multiple years to rewrite it can be beneficial to forego the high performance. A great advantage of RAJA is the involvement of the LLNL in its development. Because the LLNL actively uses RAJA for their own projects it is unlikely that development for RAJA will stop soon. With more development in the coming years, it is likely that RAJA support and performance will increase further reducing the drawbacks of using it.

References

[1] Oak Ridge National Laboratory. Frontier. https://www.olcf.ornl.gov/frontier/. Accessed: 2022-06-21.

[2] Top500. Frontier. https://www.top500.org/system/180047/. Accessed: 2022-05-29.

[3] LLNL. Arbitrary lagrangian-eulerian 3d and 2d multi-physics code. https://wci.llnl.gov/simulation/computer-codes/ale3d. Accessed: 2022-05-29.

[4] LLNL. Llnl and hpe to partner with amd on el capitan, projected as world's fastest supercomputer. https://www.llnl.gov/news/llnl-and-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer. Accessed: 2022-05-29.

[5] Raja github repositiry. https://github.com/LLNL/RAJA. Accessed: 2022-05-29.

[6] Raja portability suite: Enabling performance portable cpu and gpu hpc applications. https://computing.llnl.gov/projects/raja-managing-application-portability-next-generation-platforms. Accessed: 2022-05-29.

[7] David A. Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, and Thomas RW Scogland. Raja: Portable performance for large-scale scientific applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 71–81, 2019.

[8] Umpire: Managing heterogeneous memory resources. https://computing.llnl.gov/projects/umpire. Accessed: 2022-06-30.

[9] Raja performance suite. https://github.com/LLNL/RAJAPerf. Accessed: 2022-05-29.

[10] Kokkos github repository. https://github.com/kokkos/kokkos. Accessed: 2022-05-29.

[11] Yingda Cheng, Irene M Gamba, Fengyan Li, and Philip J Morrison. Discontinuous galerkin methods for the vlasov–maxwell equations. *SIAM Journal on Numerical Analysis*, 52(2):1017–1049, 2014.

[12] Richard Fitzpatrick. Particle-in-cell codes. https://farside.ph.utexas.edu/teaching/329/lectures/node96.html. Accessed: 2022-06-30.

[13] Openacc. https://www.openacc.org/. Accessed: 2022-06-30.

[14] Victor Artigues, Katharina Kormann, Markus Rampp, and Klaus Reuter. Evaluation of performance portability frameworks for the implementation of a particle-in-cell code, 2019.

[15] JaeHyuk Kwack, John Tramm, Colleen Bertoni, Yasaman Ghadar, Brian Homerding, Esteban Rangel, Christopher Knight, and Scott Parker. Evaluation of performance portability of applications and mini-apps across amd, intel and nvidia gpus. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 45–56, 2021.

[16] Sw4 github repository. https://github.com/geodynamics/sw4. Accessed: 2022-05-29.

[17] Intel representative teases the new ponte vecchio compute gpu for ai & hpc applications of the future. https://wccftech.com/intel-teases-next-gen-ponte-vecchio-gpu-compute-accelerator/. Accessed: 2022-06-30.

[18] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, Max P. Katz, Andrew Myers, Tan Nguyen, Andrew Nonaka, Michele Rosso, Samuel Williams, and Michael Zingale. Amrex: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software*, 4(37):1370, 2019.