On Applying Performance Portability Metrics

Daniela F. Daniel
Computer Science Division (IEC)
Aeronautics Institute of Technology (ITA)
São José dos Campos, Brazil
danieladaniel2004@gmail.com

Jairo Panetta
Computer Science Division (IEC)
Aeronautics Institute of Technology (ITA)
São José dos Campos, Brazil
jairo.panetta@gmail.com

***Abstract*—As we prepare for further technological advancement in supercomputing, the diversity of hardware architectures and parallel programming languages has increased to new levels. At the same time, extracting performance from so many architectures is even more difficult. In this context, the appearance of portable languages capable of generating executable code for multiple architectures has become a recurrent research target. We port a set of seven parallel benchmarks from SPEC ACCEL suite and a wave propagation code to one such portable language: the Kokkos C++ programming library. Using the original OpenACC versions of the eight codes, we apply a known performance portability metric on the OpenACC and Kokkos versions of those codes across a variety of hardware platforms and problem sizes. We observe that the portability metric is sensitive to the problem size. To remedy this deficiency, we propose a novel metric for performance portability, apply the proposed metric to the eight codes and discuss the results.**

***Index Terms*—Kokkos, OpenACC, performance, portability, benchmark, metrics**

I. Introduction

The plethora of architectures and programming languages pose a challenge for application developers. Many parallel clusters today have nodes with multiple computer architectures. In this scenario, applications need to perform well on these multiple architectures simultaneously to fulfill the need for cost-effectiveness and easy maintenance [1].

The task of maintaining legacy, production-level code is costly. That's one of the reasons why programmers must consider portability in advance, otherwise their application shall become restricted to the platform in which the code was created, removing the ability to run on other architectures. The solution of having multiple versions of the same code, one for each parallel machine, is clearly not effective. Furthermore, taking advantage of modern multi-core and many-core architectures is even more difficult because the optimization effort that is most effective for a particular hardware does not necessarily perform well on other architectures.

Another issue is that rewriting code is inevitable when working with specialized, non-portable languages since the target hardware is rapidly evolving. Ideally, it would be best to adopt a single, performance portable source code that performs acceptably on all target platforms, even if performance is sacrificed by narrow margins.

Under these circumstances, some parallel programming languages such as directive-based OpenACC [2] and OpenMP 4.5 (known as OpenMP Target) [3] have appeared to give the programmers alternatives for writing portable code. Another example of a modern portable language is a C++ template library called Kokkos [4]. Through template metaprogramming, Kokkos was built to add portability to user applications. In order to test the effectiveness of Kokkos in relation to OpenACC, we picked a set of benchmarks and a full application code capable of running on a range of architectures. This approach is more effective than focusing on a single large-scale application because we can stress Kokkos with a variety of kernels and thus better explore the features and abstractions that Kokkos offers.

We selected a set of benchmarks (a portion of the SPEC ACCEL 1.2 suite [5]) and a full application (a wave propagation code named *Fletcher*) as our test suite. We ported the original OpenACC version of the eight codes to Kokkos and then measured the performance of both OpenACC and Kokkos versions of all codes on two machines: *Exahost* and *Blaise*, both equipped with distinct Intel Xeon CPUs and NVIDIA GPUs.

We compare the performance portability of Kokkos and OpenACC on these codes by applying the metric described in [6]. This metric allows us to draw performance numbers specific to our set of platforms of interest. We also show how this metric behaves by varying the problem size. As expected, an application might perform well until the problem size increases to a point where the performance measures no longer hold. In a sense, these comparisons intend to reveal if problem size should also be considered when analyzing performance portability. To encompass problem size variation, we propose an alternative metric to the now popular performance portability metric from [6], apply the proposed metric to all codes in all machines and draw results.

This work is structured as follows: section 2 brings the required background and related work; in section 3 we digress to the meaning of a problem and an application that solves it; in section 4 we run the benchmarks and compute their portability; in section 5 we describe a new metric for performance portability and use it on the benchmarks; section 6 presents our experiments on *Fletcher* and in section 7 we present our conclusions and opportunities of future work.

This work has been partially supported by Petrobras under the project No. 2016/00133-9.

II. Related Work

There are many metrics for measuring productivity, programming effort, performance and, more recently, performance portability. This section presents a brief review of some studies that have proposed ways of quantifying the productivity, effort, performance, and portability of the many existing parallel programming languages. We shall visit the publications and present the metrics we deemed related to this work.

A. Empirically comparing languages

Back in 2005, studies regarding the utility of specific programming models, languages, and practices within the HPC community were rare, but researchers had already realized the importance of studying productivity and performance of HPC applications.

In that year, Hochstein et al. [7] conducted empirical studies on the human effort required to develop parallel applications, plus the amount of machine time required to execute them. These studies were conducted on graduate-level introductory HPC classrooms. Students coded two simple parallel applications ("game of life" and "grid of resistors") using both OpenMP [3] and MPI [8].

By instrumenting the students' development process, they concluded that "more effort will be required to write an MPI code than will be required to write an OpenMP code for the same application". They did not attempt to compare the performance of the two models, but rather assess whether the students were able to achieve any speedup or not. Although the students were successful in achieving speedup, there were not enough data to compare the parallelism between the two models.

B. Measuring the programming effort

In 2017, Memeti et al. [9] exploited the available performance of various parallel programming languages, namely multi-core OpenMP [3], OpenCL [10], OpenACC [2] and CUDA [11] in order to empirically study their characteristics with respect to programming productivity, performance, and energy consumption. Experiments were conducted using the industry-standard SPEC ACCEL benchmark suite [5] and the Rodinia benchmark suite [12] for accelerated computing on two heterogeneous systems that combined Intel Xeon E5 Processors with an NVIDIA GPU accelerator or an Intel Xeon Phi co-processor. Their work addressed the challenge for the developers to select one of the many available parallel programming languages according to the aforementioned criteria.

They used homegrown and third-party measurement tools to quantify the programming effort required to parallelize a code and to assess the performance and energy consumption of the benchmarks. They counted the lines of code (LOC) corresponding to parallel constructs (such as OpenMP compiler directives or CUDA specific code lines) into LOC_{par}. Likewise, the total number of LOC (legacy code plus parallel constructs) is given by LOC_{total}. Each application has its own LOC_{par} and LOC_{total}, therefore, their definition of parallelization effort is as follows:

$$\mathit{Effort}_{par}\,[\%] = 100 * LOC_{par}/LOC_{total} \tag{1}$$

Major observations from their work included: (1) OpenCL required much more effort than OpenACC for SPEC ACCEL; (2) OpenCL on average required about two times more effort than CUDA for Rodinia; (3) OpenMP required less effort than OpenCL and CUDA; (4) OpenMP, OpenCL, and CUDA had comparable performance and energy consumption but OpenCL performed better than OpenACC for SPEC ACCEL.

Notice that, up to this point, neither work had addressed portability. With today's modern languages, there's the possibility of compiling a parallel source code to different target architectures, affecting productivity and coding effort.

C. A quantitative approach for measuring portability

A few years ago, Pennycook et al. [6], [13] presented a concise definition for performance portability and an associated metric that accurately captured the performance and portability of an application across different platforms. They were aware that productivity and coding effort measures are inherently subjective and not adequate to use in order to label some language as portable or not. They examined the possibility of delivering high performance portability from a single source code and retroactively applied this metric to previous research and numerous programming languages.

To formally quantify performance portability, they defined some basic terms: a *platform* is a particular execution environment; an *application* is any suite of software that can accept a given *problem* as input and produce an output that can be validated against some existing measure of correctness; *performance* is any measurable property of an application's correct execution of a problem on a platform; *portability* is the ability of an application to execute a problem correctly on a given set of platforms. Finally, they defined *performance portability* as a measurement of an application's performance efficiency for a given problem that can be executed correctly on all platforms of a given set.

There are two metrics for performance efficiency: (1) *architectural efficiency* (achieved performance as a fraction of "peak" theoretical hardware performance); and (2) *application efficiency* (achieved performance as a fraction of best observed performance). The latter represents the ability of an application to use the most appropriate implementation for each platform and the former is the achieved performance in relation to the *roofline* model [14].

Let $e_i(a,p)$ be the efficiency of application a solving problem p on platform i. The performance portability $\mathbf{\Phi}$ metric is the harmonic mean of an application's efficiency e observed across a set of platforms H on solving problem p:

$$\mathbf{\Phi} = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,\ p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$

where $|H|$ is the cardinality of H.

Kokkos has been the subject of studies that used this metric. For example, Kirk et al. [15] investigated different implementations of *TeaLeaf*, a mini-application that solves the linear heat conduction equation. They concluded that Kokkos version was worse than the set of implementations written in OpenACC, OpenMP, CUDA and MPI.

D. Productivity and programming effort revisited

Recent studies have revisited productivity and the effort required to write HPC applications. Harrell et al. [1] developed metrics of application development productivity, which can be used to assess how productively a performance portable application was developed. They claim that the metric for performance portability in [6] does not help developers decide which among the several approaches to developing performance portable code is most effective.

They tackled the *porting* process and proposed a measure of how different the codes for solving a problem p (ported to various languages) are from each other. They called this measure *code divergence* D which is the average of the pairwise distances between the applications in the set of codes A:

$$D(A) = \binom{|A|}{2}^{-1} \sum_{\{a_i,a_j\} \subset A} d(a_i, a_j) \tag{3}$$

Note that $A \subset A_p$, where A_p is the set of all applications which solve p. The distance $d(a_i, a_j)$ between applications a_i and a_j is the change in the number of LOC normalized to the smaller (in LOC) application:

$$d(a_i, a_j) = \frac{|\text{LOC}(a_i) - \text{LOC}(a_j)|}{\min(\text{LOC}(a_i), \text{LOC}(a_j))} \tag{4}$$

The code divergence does not act as a measure of productivity, but has the potential to reveal how smooth or tough the porting effort can be. However, if this measure is sufficiently low and the application is performance portable, those numbers might characterize the adopted parallel language as the most suited.

The work of Harrell et al. goes beyond as to show metrics for development cost and developer productivity as well as to present a "methodology and associated tools for tracking application performance and portability alongside developer productivity". However, their intent on measuring performance portability is different from the path we take. In our case, instead of tracking the development process and the coding effort, our priority was to assess how the performance portability metric behaves when changing the input size of different applications and use the results to compare two portable languages.

III. Definitions

We keep the definition of *application* as being any suite of software that can accept a given *problem*. On the other hand, the definition of *problem* deserves some thought. It is well understood that an *application* solves a *problem*, but what is the difference between two applications that solve the exact same problem? Furthermore, can a problem be distinguished by its input and output data? Can a single problem accept a variety of input data and produce a variety of output data? We shall revisit some definitions and reason on how those can possibly lead to unorthodox uses of performance portability metrics.

A. What is a problem?

Our definition of problem is as follows: a *problem* is a proposed question that requires an algorithm to be solved correctly. An algorithm has the prerogative of accepting a wide range of input data and produce verifiable output data. Note that any chosen algorithm – unless the solution is hard-coded – has the capability of solving a problem for any valid input data set. For example, consider the *problem* of solving the linear system $y = Ax$, where y is an input vector and A is an input matrix. An algorithm can produce a correct output vector x for any valid A and y. This problem is not constrained by the size nor the contents of its inputs, regarded that both are valid (i.e., there's a solution for the linear system).

Note that the strategy, or the *algorithm* for solving a given *problem* is the identity of an *application*. As a consequence, two distinct applications solve the same problem if they use different algorithms. Going back to the definition in [6]: "two completely separate source codes that solve the same problem to satisfactory accuracy are the same application", unless they use different algorithms. In our example, one could solve $y = Ax$ using the conjugate gradient or an LU factorization, thus having two possible applications that solve the same problem.

B. Metrics for comparing applications

The metrics in equations 2, 3 and 4 are used to compare codes of a single application, but they can accommodate different applications that solve the same problem. We've never seen such use of the metrics, but it's important to make clear that the purpose of a metric is to quantify how effectively or productively a problem is solved; and in doing so possibly comparing different applications.

C. Problem, inputs, outputs and portability

Consider a single application that handles a problem which takes a set of inputs. This application is written in two languages, each one capable of generating code for a single platform. The results generated by those platforms compose a set of outputs. This application is also written in a portable language capable of generating code for both platforms, which then produce the desired, accurate set of outputs. Fig. 1 shows the whole process. This work is about exercising this process with two portable languages (Kokkos and OpenACC) and a set of applications in order to measure performance portability for varying problem sizes.

IV. Portable languages and benchmarks

A. Kokkos

Kokkos is a C++ template library which was developed at Sandia National Laboratories and aims to offer means for

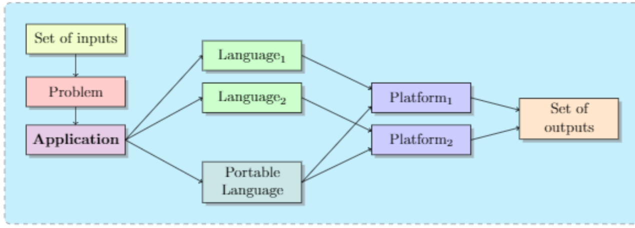

Fig. 1. An application written in different languages

scientists to write performance portable codes [4]. It requires no additional modifications to the base C++ language to work. Kokkos provides a series of abstractions that allow user code to be compiled for a variety of architectures. Kokkos has matured over the past years in that regard, specially when compiling against the supported Pthreads, OpenMP, and CUDA *backends*. If the ongoing development effort continues, Kokkos shall provide performance portability to future-generation machines, preventing developers from having to adopt yet another parallel programming language. Kokkos' website [16] delivers an overview of language features and abstractions.

B. SPEC ACCEL

The set of benchmarks chosen for this work covers intra-node parallelism with the option of offloading the computation to an accelerator (device) or simply performing the computation on a CPU (host). The SPEC ACCEL version 1.2 codes [5] are available in OpenMP Target, OpenCL and OpenACC. We chose to work with OpenACC. Out of the fifteen codes written in OpenACC, we picked seven – which are written in C – and ported them to C++ using Kokkos. The selected OpenACC benchmarks are shown in Table I.

C. Benchmarks description

Benchmarks *ep*, *cg*, *sp* and *bt* are part of NAS [17]; *Stencil*, *lbm* and *mri-q* are part of Parboil [18]. Each application is briefly described as follows:

- **Embarrassingly Parallel (EP)**: generates n pairs of random numbers which are uniformly distributed on the interval $(-1, 1)$. Each pair is tested by Marsaglia's polar method [19]. It's an acceptance-rejection procedure that works by testing random pairs (x_j, y_j) to check if $t_j = x_j^2 + y_j^2 \leq 1$, for $1 \leq j \leq n$. If this inequality holds, then it can be proved that $X_k = x_j\sqrt{(-2\ln t_j)/t_j}$ and $Y_k = y_j\sqrt{(-2\ln t_j)/t_j}$ are independent Gaussian deviates with mean zero and variance one. The code keeps a tally of all accepted pairs (x_j, y_j) that lie in the square annulus $l \leq \max(|X_k|, |Y_k|) < l+1$ for $0 \leq l \leq 9$ and computes the sum of all deviates generated;
- **Conjugate Gradient (CG)**: is an iterative method for solving sparse systems of linear equations [20]. This benchmark is used to compute an approximation to the smallest eigenvalue of the sparse, symmetric positive definite matrix A. It solves a linear system of the form $Az = x$ and uses the inverse power method [21] to find an estimate of the eigenvalue of A;
- **Scalar Penta-diagonal (SP)**: let u and r be 5x1 vectors defined at the points of a 3D rectangular grid and K be a 7-diagonal block matrix of 5x5 blocks. SP solves a 3D discretization of Navier-Stokes equations: $Ku = x$. SP uses the Beam-Warming approximate factorization of K:

$$K \cong T_x * P_x * T_x^{-1} * T_y * P_y * T_y^{-1} * T_z * P_z * T_z^{-1}$$

  T_x, T_y and T_z are block diagonal matrices of 5x5 blocks; P_x, P_y and P_z are scalar penta-diagonal matrices. The resulting system is then solved by inverting the block diagonal matrices T_x, $T_x^{-1} \times T_y$, $T_y^{-1} \times T_z$ and T_z^{-1} and then solving the scalar penta-diagonal systems [22];
- **Block Tridiagonal (BT)**: differs from SP in the factoring of K. It uses Alternating Direction Implicit (ADI) approximate factorization of K that decouples the x, y and z dimensions:

$$K \cong BT_x * BT_y * BT_z$$

  BT_x, BT_y and BT_z are block tridiagonal matrices of 5x5 blocks. The resulting system is then solved by computing the block tridiagonal systems in x, y and z directions successively [22];
- **Stencil**: solves a simple 3D heat equation by sweeping over a spacial grid. The accelerated loop is a fairly simple 7-point stencil operation;
- **Lattice Boltzmann Method (LBM)**: is a technique for solving the Navier-Stokes equations based on a microscopic model of the moving fluid particles [23]. The entire grid (lattice) is traversed and each cell uses the input flows at the previous discrete point in time to compute the resulting output flows from that cell and an updated local fluid density (in practice, it's a 19-point stencil);
- **Magnetic Resonance Imaging (MRI-Q)**: computes part of the advanced reconstruction algorithm described in [24], i.e., the computation of each element of matrix Q, given by:

$$Q(X_n) = \sum_{m=1}^{M} |\phi(k_m)|^2 e^{(i2\pi k_m \cdot x_n)}$$

  where "·" denotes the inner product. There are M K-space sampling locations, with k_m denoting the location

TABLE I
OPENACC BENCHMARKS

Application	Application Domain
ep	Random number generation
cg	Conjugate gradient
sp	Scalar pentadiagonal solver
bt	Block tridiagonal solver for 3D PDE
stencil	Thermodynamics
lbm	Lattice Boltzmann method
mri-q	Medicine

of the mth sample. Likewise, there are N voxel (volumetric pixel) coordinates, with x_n denoting the coordinates of the nth voxel.

D. Measuring performance portability

We ran the benchmarks on a Linux Ubuntu 18.04.2 LTS on two different machines equipped with Intel CPUs and NVIDIA GPUs. The first machine – *Exahost* – has a dual-socket Intel Xeon E5-2630 v4 and a Kepler generation NVIDIA Tesla K80. The second machine – *Blaise* – has a dual-socket Intel Xeon E5-2699 v4 and a Pascal generation NVIDIA Tesla P100 GPU. The ported codes to Kokkos use OpenMP *backend* when running on CPUs and CUDA *backend* when running on GPUs. There are no tweaks of any portable code for a specific platform.

We measured the execution time (see appendix B) of each benchmark for a variety of problem sizes[1] in both OpenACC and Kokkos versions. Reported execution times are the mean of three runs on each platform and have been rounded to two decimal places for aesthetics. Tables II, III, IV and V report the application efficiency computed with these execution times. We used the application efficiency exclusively for calculating performance portability; the architectural efficiency is harder to obtain and wouldn't be of much use in comparing the two languages because even the best known performing application has low architectural efficiency [6]. In our experiments, application efficiency offers a practical upper bound, and it is easily computed based on direct measurements [25]. Table VI brings resulting performance portability numbers galore, considering all four platforms.

E. Discussion

This handful of data shall not be overlooked. We present our conclusions for every benchmark:

- **EP**: Kokkos performed better than OpenACC on the CPUs and on the K80 (except for size 1), but OpenACC wins on the P100. Kokkos is the best choice, since Φ values are above 90% for Kokkos and 80% for OpenACC;
- **CG**: Kokkos has the best performance on the GPUs (except for size 5) and was better than OpenACC for the smaller sizes (1 and 2) on the CPUs. Observe that OpenACC becomes more efficient and thus performance portable as the problem size increases. OpenACC Φ values range from 72% to 100%, reaching over 90% for sizes 3, 4 and 5;
- **SP**: OpenACC prevails over Kokkos on all platforms. Consequently, Table VI shows straight 100% values for OpenACC on all input sizes, while the best performance portability of Kokkos is no better than 40%. Furthermore, Kokkos performance portability decreases as problem size increases;
- **BT**: Kokkos was more consistent on the CPUs, but OpenACC is best overall. Kokkos surpassed OpenACC running with inputs 1 and 2 on Xeon E5-2630. Poor Φ results are due to poor performance on the GPUs. Once again, Kokkos' numbers grew worse as the problem size increased;

[1]Refer to appendix A for information on compilers and problem sizes.

TABLE II
APPLICATION EFFICIENCY ON XEON E5-2630 V4

	App.	Input Size				
		1	2	3	4	5
OpenACC	*ep*	80.75%	74.86%	83.33%	77.76%	78.84%
	cg	97.29%	85.69%	100%	100%	100%
	sp	100%	100%	100%	100%	
	bt	92.23%	71.85%	100%		
	stencil	30.91%	58.37%	78.42%	100%	100%
	lbm	82.72%	97.10%			
	mri-q	100%	100%			
Kokkos	*ep*	100%	100%	100%	100%	100%
	cg	100%	100%	91.95%	95.69%	95.78%
	sp	64.83%	58.89%	69.12%	66.66%	
	bt	100%	100%	99.25%		
	stencil	100%	100%	100%	72.32%	57.90%
	lbm	100%	100%			
	mri-q	46.90%	29.06%			

TABLE III
APPLICATION EFFICIENCY ON XEON E5-2699 V4

	App.	Input Size				
		1	2	3	4	5
OpenACC	*ep*	82.31%	82.04%	80.24%	77.60%	78.28%
	cg	99.16%	94.83%	100%	100%	100%
	sp	100%	100%	100%	100%	
	bt	100%	100%	100%		
	stencil	100%	100%	100%	100%	100%
	lbm	71.91%	98.61%			
	mri-q	100%	100%			
Kokkos	*ep*	100%	100%	100%	100%	100%
	cg	100%	100%	93.89%	86.30%	98.51%
	sp	33.11%	32.37%	48.91%	71.97%	
	bt	35.29%	70.52%	72.99%		
	stencil	78.19%	87.97%	69.91%	59.10%	71.52%
	lbm	100%	100%			
	mri-q	81.86%	37.93%			

TABLE IV
APPLICATION EFFICIENCY ON TESLA K80

	App.	Input Size				
		1	2	3	4	5
OpenACC	*ep*	100%	83.63%	78.55%	77.47%	76.82%
	cg	76.87%	89.90%	95.71%	99.82%	100%
	sp	100%	100%	100%	100%	
	bt	100%	100%	100%		
	stencil	83.05%	100%	100%	100%	100%
	lbm	99.52%	100%			
	mri-q	83.51%	99.41%			
Kokkos	*ep*	98.02%	100%	100%	100%	100%
	cg	100%	100%	100%	100%	98.23%
	sp	27.35%	21.47%	11.93%	11.15%	
	bt	10.38%	10.99%	7.80%		
	stencil	100%	91.69%	80.23%	60.43%	58.81%
	lbm	100%	98.28%			
	mri-q	100%	100%			

TABLE V
APPLICATION EFFICIENCY ON TESLA P100

	App.	Input Size 1	2	3	4	5
OpenACC	*ep*	100%	100%	100%	100%	100%
	cg	45.59%	51.34%	73.43%	89.82%	100%
	sp	100%	100%	100%	100%	
	bt	100%	100%	100%		
	stencil	100%	100%	100%	100%	100%
	lbm	98.32%	100%			
	mri-q	100%	98.70%			
Kokkos	*ep*	85.67%	78.92%	77.19%	76.92%	76.25%
	cg	100%	100%	100%	100%	96.75%
	sp	63.74%	43.25%	28.56%	23.36%	
	bt	17.88%	10.49%	9.98%		
	stencil	86.10%	79.26%	76.94%	59.20%	60.59%
	lbm	100%	90.91%			
	mri-q	93.28%	100%			

TABLE VI
PERFORMANCE PORTABILITY $\mathbf{\Phi}$

	App.	Input Size 1	2	3	4	5
OpenACC	*ep*	89.82%	84.20%	84.76%	82.21%	82.52%
	cg	72.32%	75.73%	90.77%	97.20%	100%
	sp	100%	100%	100%	100%	
	bt	97.94%	91.08%	100%		
	stencil	62.12%	84.87%	93.56%	100%	100%
	lbm	86.56%	98.91%			
	mri-q	95.30%	99.52%			
Kokkos	*ep*	95.53%	93.74%	93.12%	93.02%	92.78%
	cg	100%	100%	96.32%	95.15%	97.30%
	sp	40.87%	34.02%	26.01%	24.79%	
	bt	20.98%	19.00%	15.86%		
	stencil	90.08%	89.11%	80.38%	62.32%	61.77%
	lbm	100%	97.15%			
	mri-q	73.72%	49.52%			

- **Stencil**: OpenACC performed better on three platforms: the GPUs and the Xeon E5-2699. Kokkos' efficiency also decreases with increasing input: we noticed this is sort of a trend. $\mathbf{\Phi}$ measures indicate that OpenACC prevails;
- **LBM**: Kokkos and OpenACC showed competitive performance. Kokkos was a bit better on the CPUs and the numbers on the GPUs are tight. Portability measures slightly favor Kokkos;
- **MRI-Q**: Kokkos is significantly slower on the CPUs, but competitive on the GPUs. Portability results put OpenACC as the best choice. Poor results on Xeon E5-2630 ended up spoiling Kokkos' portability.

V. ANOTHER METRIC FOR PERFORMANCE PORTABILITY

It should be evident by now that performance portability depends on the problem size. This was observed in all the benchmarks tested, and that is a strong evidence that this metric is sensitive to the input size. The work of Pennycook et al. [6] does not address how to appropriately use the metric when there is a set of inputs for the same application.

We do not find a whole set of $\mathbf{\Phi}$ percentages for the same problem useful for characterizing how effective a portable parallel programming language is at solving that problem. This situation motivated us to make an attempt at creating a novel metric for performance portability which is capable of offering a single measure for an application stressed by a number of input sizes and configurations.

A. Shrinking performance portability measures

We adopt the very same definitions of *platform*, *performance* and *portability* as in [6], but use our own definition (see section III) of *application*. The two functions of performance efficiency (*architectural* and *application* efficiency) proposed in [6] are equally valuable because both efficiency functions grant the user freedom to select the most appropriate performance evaluation method. We preserve this flexible aspect of performance evaluation for our metric. The work in [25] offers detailed guidance on different ways of calculating performance.

Since we are comparing performance portable codes, the idea of code divergence in equation 4 comes in handy because it compares two implementations of the same problem written in two different languages [1]. We replace the LOC measure with a performance metric chosen by the user. That way, instead of directly computing the architectural or application efficiency, we calculate the *relative error* in relation to the best observed or theoretical hardware performance.

Consider the set A_p of applications that solve problem p; let $f_i(a, p, s)$ be a measure of performance of application a on solving p on platform i with input size s. We define the *performance distance* $\delta(a, \alpha)$ between applications a and α as the relative error in performance. Note that $f_i(\alpha, p, s)$ is the best observed result or the theoretical hardware performance:

$$\delta(a, \alpha) = \frac{|f_i(a, p, s) - f_i(\alpha, p, s)|}{f_i(\alpha, p, s)} \quad (5)$$

We call *Divergence RMS* Δ_{RMS} the *root mean square* of performance distances between a set of input sizes of interest S:

$$\Delta_{RMS} = \sqrt{\frac{\sum_{s \in S} \delta(a, \alpha)^2}{|S|}} \quad (6)$$

The root mean square of distances measures the average size of its entries, i.e., the typical performance distance of the set of inputs. The RMS is equal to or higher than the arithmetic mean and that gives more "significance" to the performance distances obtained. Finally, we define the *Performance Portability Divergence* $\mathcal{P}_{\mathcal{D}}$ as the arithmetic mean of RMS divergences across a set of platforms H:

$$\mathcal{P}_{\mathcal{D}} = \frac{\sum_{i \in H} \Delta_{RMS}}{|H|} \quad (7)$$

An obvious thing to notice is that the ideal performance portability divergence should be zero and the perfect performance portability is 100%. We shall use data on the benchmarks to demonstrate how to compute and interpret the performance portability divergence of OpenACC and Kokkos. We also reason on the implications of using this metric and try to expose its limitations. As proposed, our results offer a

single measure for estimating the performance portability of an application over a variety of input data.

B. Performance Portability Divergence of the benchmarks

Table VII shows the Δ_{RMS} and $\mathcal{P}_{\mathcal{D}}$ values of the benchmarks calculated with the execution times shown in appendix B. Kokkos is the best option for *ep*, *cg*, and *lbm*; OpenACC leads Kokkos on *stencil*, *mri-q*, and is a far better option for *sp* and *bt*.

$\mathcal{P}_{\mathcal{D}}$ values are somewhat "compliant" with the numbers in Table VI, i.e., they are not conflicting measures. Poor values of performance portability end up reflecting on bad performance portability divergence and the other way around is also true.

However, $\mathcal{P}_{\mathcal{D}}$ is more "pessimistic" because it tends to punish large performance distances severely. This is the effect of having an RMS value as Δ; a simple arithmetic mean would provide lower values. The rationale for penalizing large performance distances comes from the fact that a portable, production-level application should be versatile enough to accept a myriad of input data and this has to be accounted for in our metric. Finally, since all four platforms are equally important, the arithmetic mean of Δ_{RMS} values produces a unique $\mathcal{P}_{\mathcal{D}}$ across the set H.

C. Discussion

There are two types of performance functions: "higher is better" and "lower is better". A consequence of applying the latter is that the distance is not normalized, i.e., could be higher than 100%. The execution time falls into this category; we reached $\mathcal{P}_{\mathcal{D}}$ values greater than 100% with *bt* and *sp* in Kokkos. Results suggest that it is common to reach over 40% $\mathcal{P}_{\mathcal{D}}$ (4 cases out of 7). The numbers support the "harsh" property of $\mathcal{P}_{\mathcal{D}}$: it is more challenging to reach nearly 0% for $\mathcal{P}_{\mathcal{D}}$ than it is to reach 100% for performance portability.

Similarly to the portability metric in [6], poor measures are a strong evidence of the potential left for optimization. We believe that both metrics offer valuable help in providing a quantitative measure of performance portability useful for deciding which parallel language is most suited for the users' needs and at the same time worthy of optimization efforts. For example, *bt* and *sp* didn't perform well on Kokkos, as caught by $\mathcal{P}_{\mathcal{D}}$.

As for the remaining benchmarks, Kokkos was best at running *ep*, *cg* and *lbm*, whose $\mathcal{P}_{\mathcal{D}}$ stayed below 10%. In *mri-q*, OpenACC outperformed Kokkos on the CPUs, compromising both $\mathbf{\Phi}$ and $\mathcal{P}_{\mathcal{D}}$. OpenACC also has an edge in *stencil*, but its $\mathcal{P}_{\mathcal{D}}$ is far from satisfactory.

TABLE VII
Δ_{RMS} AND $\mathcal{P}_{\mathcal{D}}$ OF THE BENCHMARKS

	App.	Δ_{RMS}				$\mathcal{P}_{\mathcal{D}}$
		K80	P100	E5-2630	E5-2699	
OpenACC	*ep*	24.02%	0%	26.96%	25.10%	19.02%
	cg	14.51%	70.24%	7.57%	2.47%	23.70%
	sp	0%	0%	0%	0%	0%
	bt	0%	0%	23.14%	0%	5.78%
	stencil	9.13%	0%	105%	0%	28.69%
	lbm	0.34%	1.21%	14.92%	27.63%	11.02%
	mri-q	13.97%	0.93%	0%	0%	3.73%
Kokkos	*ep*	0.90%	27.34%	0%	0%	7.06%
	cg	0.81%	1.50%	4.82%	7.70%	3.71%
	sp	588%	218%	55.48%	155%	254%
	bt	966%	764%	0.44%	110%	460%
	stencil	44.46%	46.53%	36.74%	42.88%	42.65%
	lbm	1.24%	7.07%	0%	0%	2.08%
	mri-q	0%	5.09%	190%	116%	78.04%

VI. A FULL APPLICATION

In order to exercise the metrics on a code other than a benchmark, we elected a production-level wave propagation code named *Fletcher* as our real-life example. This application is a cornerstone of the oil and gas industry. It simulates the propagation of acoustic waves over time on a 3D anisotropic media. Anisotropy is limited by a set of tilted transversely isotropic (TTI) planes. Wave propagation is modeled by coupled pseudo-acoustic PDEs as suggested by Fletcher, Du and Fowler [26]. Denoting pressure by $p(x, y, z, t)$ and using the auxiliary variable $q(x, y, z, t)$, the PDEs are

$$\begin{aligned} \frac{\partial^2 p}{\partial t^2} &= v_{px}^2 H_2 p + \alpha v_{pz}^2 H_1 q + v_{sz}^2 H_1 (p - \alpha q) \\ \frac{\partial^2 q}{\partial t^2} &= \frac{v_{pn}^2}{\alpha} H_2 p + v_{pz}^2 H_1 q - v_{sz}^2 H_2 (\frac{1}{\alpha} p - q) \end{aligned} \tag{8}$$

where $v_{px}(x, y, z)$, $v_{pz}(x, y, z)$ and $v_{sz}(x, y, z)$ are directional propagation speeds, α is a coupling parameter and the differential operators H_1 e H_2 are defined by:

$$\begin{aligned} H_1 =& sin^2\theta cos^2\phi \frac{\partial^2}{\partial x^2} + sin^2\theta sin^2\phi \frac{\partial^2}{\partial y^2} + \\ & cos^2\theta \frac{\partial^2}{\partial z^2} + sin^2\theta sin2\phi \frac{\partial^2}{\partial x \partial y} + \\ & sin2\theta sin\phi \frac{\partial^2}{\partial y \partial z} + sin2\theta cos\phi \frac{\partial^2}{\partial x \partial z} \\ H_2 =& \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} - H_1 \end{aligned} \tag{9}$$

where θ and ϕ are TTI's dip and azimuth angles and α is a coupling parameter. Discretization is based on finite differences, approximating space derivatives by operators of order eight and time derivatives by second order ones. The original program was written in C and parallelized by OpenACC.

Five domain sizes have been tested on *Exahost* and *Blaise*. The achieved performance (in GFlop/s) on each platform is shown in Fig. 2 and Fig. 3. We remind the reader that this work isn't about optimizing code; we expect improvements in Kokkos' numbers at later stages of development. Clearly, OpenACC outperforms Kokkos in all cases, which gives us a straight 100% application efficiency and $\mathbf{\Phi}$ for every media size. Likewise, all divergences are 0%.

Kokkos' $\mathbf{\Phi}$ numbers, however, range from 31% to 40% as shown in Table VIII, demonstrating that $\mathbf{\Phi}$ does change with problem size, even though Kokkos never dethroned OpenACC as best observed. Kokkos has reached distinct Δ_{RMS} values across platforms: 71.32% on Xeon E5-2630; 74.32% on Xeon

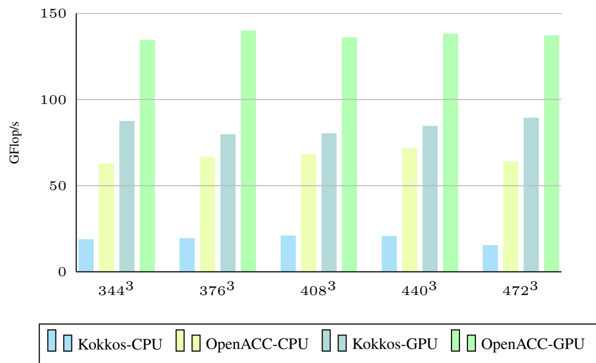

Fig. 2. Performance running on *Exahost*

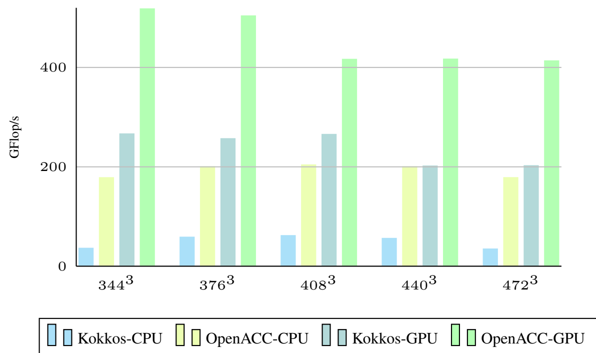

Fig. 3. Performance running on *Blaise*

E5-2699; 38.62% on K80 and 47.55% on P100. The performance portability divergence $\mathcal{P}_{\mathcal{D}}$ of Kokkos is **57.96%**.

Note that, as apposite to the benchmarks, the performance of *Fletcher* on a single platform does not change much with problem size. This is a particular case that has an impact on $\mathcal{P}_{\mathcal{D}}$ by producing a smoother value than $\mathbf{\Phi}$. Nonetheless, Kokkos would still have to improve on every problem size to lower its $\mathcal{P}_{\mathcal{D}}$ values. It seems easier on *Fletcher* but not so on size-sensitive applications like the benchmarks.

TABLE VIII
PERFORMANCE PORTABILITY OF KOKKOS

	TTI media size				
	344^3	376^3	408^3	440^3	472^3
$\mathbf{\Phi}$	34.41%	38.06%	40.93%	37.50%	31.45%

VII. CONCLUSIONS AND FUTURE WORK

The pursuit of performance portability and ways of measuring it has motivated studies that try to measure performance, portability, and productivity, or directly make a comparison of two or more portable languages. We've run seven benchmarks and a full application written in both Kokkos and OpenACC. That is, two portable languages that can compile down to the hardware platforms present in this work: Intel Xeon CPUs and NVIDIA GPUs. In all experiments, performance portability measures changed with input size, meaning that the best performing portable language for a given application might change depending on the input size.

In order to unify varying values of $\mathbf{\Phi}$, we proposed a novel metric called Performance Portability Divergence – denoted by $\mathcal{P}_{\mathcal{D}}$ – which takes into consideration a user-defined set of input sizes to quantify the portability of an application. As opposed to $\mathbf{\Phi}$, the perfect desirable value of $\mathcal{P}_{\mathcal{D}}$ is 0%. From the beginning, measures of $\mathbf{\Phi}$ and $\mathcal{P}_{\mathcal{D}}$ were supposed to "agree" since the application of both metrics shall rigorously use a similar measure of performance (e.g., execution time, Flop/s, bandwidth, etc).

We concluded that selecting the best language for a given application is harder when dealing with varying problem sizes. We do believe that real-life HPC applications are meant to work effectively for diverse input sizes. The employed metrics will probably show biased results if the set of inputs is over or under-representative of actual workloads. We also concluded that performance portability divergence is extremely sensible to the set of inputs chosen. We advise considering only the relevant inputs of a given application. For example, an application may perform poorly for a particular input size, causing the performance portability divergence to increase, regardless of the results from other input sizes.

$\mathcal{P}_{\mathcal{D}}$ is susceptible to the same criticism as $\mathbf{\Phi}$ when distances are computed in relation to the best observed performance; if the best implementation is poor, it is easier to get better results. The user has the option to use the *roofline* model, but this typically provides low efficiency measures as seen in [9], making it difficult to compare the languages.

We find important to mention that there hasn't been any attempt to improve the performance portability of Kokkos through optimization. The parallel kernels are as similar as possible to the original version in OpenACC. We are aware that Kokkos allows use of more advanced features, which were kept untouched in our work.

A. Future work

A strong reason to extend this work is to try and benefit from the fact of having a real-life problem to evaluate (*Fletcher*, in our case) and change additional parameters such as the method for measuring performance (use the *roofline* model) or confront highly optimized portable implementations with state-of-the-art – commonly non-portable – legacy applications, therefore setting the "best observed" measurement to the uppermost level.

Optimizing the Kokkos version of *Fletcher* would be a natural starting line for testing the behavior of our metric. As seen previously, OpenACC showed good performance, but there are fine-tuned, heavy-duty versions of *Fletcher* written in other languages.

This leads to another approach for future research that consists of expanding the set of portable languages being analyzed. As compilers and languages mature fast, more portable applications which solve the same problem shall surface, exposing the relevance of having a solid, fair and meaningful methodology for comparison.

VIII. Acknowledgements

The authors wish to thank Pedro Pais Lopes for providing the OpenACC version of *Fletcher*.

References

[1] S. L. Harrell, J. Kitsonz, R. Bird, S. J. Pennycook, J. Sewall, D. Jacobsen, D. N. Asanza, A. Hsu, H. C. Cabada, H. Kim, and R. Robey, "Effective Performance Portability," *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, Dallas, TX, USA, 2018, pp. 24-36.

[2] OpenACC. [Online]. Available: https://www.openacc.org/

[3] The OpenMP API specification for parallel programming. [Online]. Available: https://www.openmp.org/

[4] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 32023216, 2014.

[5] G. Juckeland, W. Brantley, S. Chandrasekaran, B. Chapman, S. Che, M. Colgrove, H. Feng, A. Grund, R. Henschel, W.-M. W. Hwu, H. Li, M. S. Mller, W. E. Nagel, M. Perminov, P. Shelepugin, K. Skadron, J. Stratton, A. Titov, K. Wang, M. V. Waveren, B. Whitney, S. Wienke, R. Xu, and K. Kumaran, "SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance," *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, New Orleans, LA, USA, 2014, pp. 46–67.

[6] S. J. Pennycook, J. Sewall, and V. Lee, "A metric for performance portability," *arXiv preprint arXiv:1611.07409*, 2016.

[7] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, and M. V. Zelkowitz, "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers," *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, Seattle, WA, USA, 2005, pp. 35-35.

[8] Open MPI. [Online]. Available: https://www.open-mpi.org/

[9] S. Memeti, L. Li, S. Pllana, J. Kolodziej, and C. Kessler, "Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming productivity, performance, and energy consumption," *arXiv preprint arXiv:1704.05316v1*, 2017.

[10] Khronos OpenCL Working Group, *The OpenCL Specification Version 2.2*, March 2016.

[11] CUDA Toolkit Documentation. CUDA C Programming Guide [Online]. Available: docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[12] S. Che, M. Boyer, J.Meng, D. Tarjan, W. J. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, 2009, pp. 44-54.

[13] S. J. Pennycook, J. Sewall, and V. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, vol. 92, pp. 947958, 2019.

[14] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no.4, pp. 6576, 2009.

[15] R. O. Kirk, G. R. Mudalige, I. Z. Reguly, S. A. Wright, M. J. Martineau and S. A. Jarvis, "Achieving Performance Portability for a Heat Conduction Solver Mini-Application on Modern Multi-core Systems," *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Honolulu, HI, 2017, pp. 834-841.

[16] Kokkos C++ Performance Portability Programming EcoSystem. [Online]. Available: https://github.com/kokkos

[17] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS Parallel Benchmarks," NASA Ames Research Center, Moffett Field, CA - USA, Tech. Rep., March 1994.

[18] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W-M. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," Center for Reliable and High-Petformance Computing, 2012.

[19] D. E. Knuth, *The art of computer programming, vol. 2*. Massachusetts: Addison-wesley, 1998.

[20] J. R. Shewchuk, *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Pittsburgh: Carnegie-Mellon University. Department of Computer Science, 1994.

[21] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia: SIAM, 2009.

[22] M. Frumkin, H. Jin, and J. Yan, "Implementation of NAS parallel benchmarks in high performance fortran," NASA Ames Research Center, Moffett Field, CA - USA, Tech. Rep., September, 1998.

[23] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, and U. Rde, "Optimization And Profiling Of The Cache Performance Of Parallel Lattice Boltzmann Codes," *Parallel Processing Letters*, vol. 13, no. 04, pp. 549560, 2003.

[24] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-M. W. Hwu, Z.-P. Liang, and B. P. Sutton, "Accelerating Advanced MRI Reconstructions on GPUs," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 13071318, 2008.

[25] H. Dreuning, R. Heirman, and A. L. Varbanescu, "A beginners guide to estimating and improving performance portability," *3rd International Workshop on Performance Portable Programming Models for Accelerators (P3MA) at the International Supercomputing Conference (ISC)*, 2018.

[26] R. P. Fletcher, X. Du and P. J. Fowler, "Reverse time migration in tilted transversely isotropic (TTI) media", *Geophysics*, vol. 74, no. 6, pp. 179-187, 2009.

Appendix A
Compilers, benchmarks problem sizes and parameters

Table IX shows the compilers and flags used to run the applications on each platform. We worked with Kokkos 2.8.

TABLE IX
COMPILERS AND FLAGS

	Compiler	Platform	Main flags
OpenACC	PGI 18.10	E5-2630	`-O3 -acc -ta=multicore`
		E5-2699	
		Tesla K80	`-O3 -acc -ta=tesla:cc35`
		Tesla P100	`-O3 -acc -ta=tesla:cc60`
Kokkos	GCC 7.3	E5-2630	`-O3 -march=native`
	GCC 6.4	E5-2699	
	CUDA 9.2.88	Tesla K80	
	CUDA 10.0	Tesla P100	

A. Sizes and configurations

The five input sizes seen in benchmarks *ep*, *cg*, *sp* and *bt* correspond to classes W to D, respectively. Details of each class can be found in *www.nas.nasa.gov/publications/npb.html*. The classes we used for *sp* were W, A, B and C and for *bt* we used classes W, A and B.

We run *stencil* with the same number of iterations (100), varying the grid sizes: $128^2 \times 64$, $256^2 \times 32$, $256^2 \times 64$, $512^2 \times 32$ and $512^2 \times 64$. Input sizes of *mri-q* are: 32^3 pixels with 3072 samples, and 64^3 pixels with 2048 samples. Both inputs of *lbm* are $100^2 \times 130$ grids, only the number of iterations (300 and 5000) and boundary conditions change. In the case of *lbm*, and *mri-q*, we used the input files from SPEC ACCEL; *stencil* grid values are generated randomly.

Appendix B
Benchmarks execution times

TABLE X
EXECUTION TIMES ON XEON E5-2630 V4 (SECONDS)

	App.	Input Size 1	2	3	4	5
OpenACC	*ep*	0.18	1.45	5.00	19.91	302.99
	cg	0.06	0.10	7.73	18.57	500.90
	sp	0.72	4.60	28.41	122.55	
	bt	3.90	88.41	357.72		
	stencil	0.25	0.29	0.37	0.51	0.81
	lbm	7.22	99.10			
	mri-q	0.04	0.12			
Kokkos	*ep*	0.15	1.09	4.17	15.49	238.88
	cg	0.05	0.08	8.41	19.41	522.95
	sp	1.11	7.81	41.11	183.84	
	bt	3.60	63.52	360.43		
	stencil	0.08	0.17	0.29	0.71	1.40
	lbm	5.97	96.22			
	mri-q	0.08	0.40			

TABLE XI
EXECUTION TIMES ON XEON E5-2699 V4 (SECONDS)

	App.	Input Size 1	2	3	4	5
OpenACC	*ep*	0.10	0.64	2.22	8.57	132.23
	cg	0.08	0.08	4.99	11.32	247.02
	sp	0.41	1.67	14.89	81.25	
	bt	1.13	28.41	134.05		
	stencil	0.05	0.08	0.11	0.22	0.52
	lbm	5.66	65.98			
	mri-q	0.03	0.07			
Kokkos	*ep*	0.08	0.53	1.78	6.65	103.51
	cg	0.08	0.07	5.32	13.11	250.77
	sp	1.23	5.16	30.44	112.89	
	bt	3.21	40.28	183.64		
	stencil	0.06	0.09	0.16	0.37	0.73
	lbm	4.07	65.07			
	mri-q	0.04	0.19			

TABLE XII
EXECUTION TIMES ON TESLA K80 (SECONDS)

	App.	Input Size 1	2	3	4	5
OpenACC	*ep*	0.37	1.54	5.62	21.45	342.91
	cg	0.16	0.28	7.72	18.99	929.52
	sp	1.54	4.76	20.35	87.55	
	bt	0.82	9.84	39.19		
	stencil	0.25	0.29	0.37	0.52	0.81
	lbm	11.92	192.79			
	mri-q	0.15	0.17			
Kokkos	*ep*	0.38	1.29	4.41	16.62	263.44
	cg	0.12	0.26	7.38	18.96	946.27
	sp	5.62	22.19	170.60	785.06	
	bt	7.87	89.56	502.63		
	stencil	0.21	0.31	0.46	0.86	1.37
	lbm	11.86	196.17			
	mri-q	0.12	0.17			

TABLE XIII
EXECUTION TIMES ON TESLA P100 (SECONDS)

	App.	Input Size 1	2	3	4	5
OpenACC	*ep*	0.12	0.34	1.05	3.84	59.96
	cg	0.07	0.10	1.79	3.70	265.48
	sp	0.84	2.08	6.79	27.90	
	bt	0.25	2.85	12.46		
	stencil	0.30	0.32	0.37	0.45	0.63
	lbm	2.82	40.52			
	mri-q	0.28	0.32			
Kokkos	*ep*	0.14	0.43	1.37	5.00	78.63
	cg	0.03	0.05	1.32	3.32	274.40
	sp	1.31	4.81	23.76	119.42	
	bt	1.41	27.14	124.89		
	stencil	0.35	0.41	0.48	0.77	1.04
	lbm	2.77	44.57			
	mri-q	0.30	0.31			