ChapelPerf: A Performance Suite for Chapel

Ricardo Jesus
rjj@ed.ac.uk
EPCC, The University of Edinburgh
Edinburgh, United Kingdom

Michèle Weiland
m.weiland@epcc.ed.ac.uk
EPCC, The University of Edinburgh
Edinburgh, United Kingdom

ABSTRACT

We present ChapelPerf, the Chapel Performance Suite. ChapelPerf is a port of RAJAPerf, the RAJA Performance Suite, to Chapel. It enables the performance of Chapel to be compared to a variety of programming models, such as C++, OpenMP, RAJA, among others. We discuss our porting experience, identifying and describing the main struggles we faced. Additionally, we present early results comparing the performance of baseline sequential and parallel implementations of the RAJAPerf kernels in Chapel to the performance of reference C++ and OpenMP implementations.

1 INTRODUCTION

Application porting and performance optimisation have always been challenging endeavours in high-performance computing. To make matters worse, the increasing number of hardware options, architectures, and overall nuances at all system scales, now more than ever make developing high-performance code that runs efficiently on a large number of hardware configurations a herculean task.

To cope with the complexity of this task, over the years, several high-level parallel programming languages, libraries, and runtimes were created. One of these languages is Chapel, a programming language designed for productive parallel computing at any system scale. Chapel aspires not only to be a highly productive language the likes of Python or MATLAB, but also to do so whilst remaining competitive performance-wise with more traditional parallel programming approaches such as those based on C/C++, OpenMP, and/or MPI.

In order to facilitate the process of assessing the performance of Chapel compared to that of alternative programming models, we ported the RAJA Performance Suite (RAJAPerf) [2] to Chapel. RAJAPerf is a collection of loop-based computational kernels commonly found in HPC codes, each of which is implemented in a variety of parallel programming models, such as C++, OpenMP, RAJA, and CUDA. The result of our port is ChapelPerf, the Chapel Performance Suite [4].

ChapelPerf is an implementation of the RAJAPerf kernels in Chapel. It serves as a drop-in replacement for RAJAPerf, enabling the reference implementations of the latter to be directly compared to Chapel implementations. Despite being in its early stages, we hope ChapelPerf may be used to (i) evaluate the performance of Chapel compared to alternative parallel programming models, (ii) assess the performance of Chapel on different system architectures, and (iii) help direct optimisation efforts to make Chapel increasingly competitive on the HPC scene.

The remainder of this paper is organised as follows. In Sec. 2 we provide a brief overview of RAJAPerf and ChapelPerf. In Sec. 3 we discuss our porting experience, pointing out some of the struggles we encountered along the way. In Sec. 4 we present and discuss some early results comparing the performance of baseline sequential and parallel implementations of the RAJAPerf kernels in Chapel to reference implementations in C++ and OpenMP, on three distinct HPE systems. Finally, in Sec. 5 we provide some concluding remarks and point out future directions for ChapelPerf.

2 BENCHMARK DESCRIPTION

The Chapel Performance Suite (ChapelPerf) is an implementation of the RAJA Performance Suite (RAJAPerf) [2] in Chapel. In this section we provide a brief description of the two benchmark suites and describe how they are related.

2.1 RAJA Performance Suite

The RAJA Performance Suite (RAJAPerf) is a benchmark suite designed to assess the performance of various parallel programming models and runtimes on a wide range of commonly-occurring loop-based kernels found in HPC codes [2]. In total, the latest release of the suite, v0.11.0[1], contains 53 kernels written in a combination of C++ 11 and other (parallel) libraries and runtimes. The kernels are borrowed from a variety of sources, such as HPC applications and other benchmark suites. The kernels are partitioned into six different groups—Algorithm, Apps, Basic, Lcals, Polybench, and Stream—, where the group name alludes to the origin and/or algorithmic patterns of its kernels. As an example, the "Apps" group contains a collection of kernels extracted from real-world HPC applications, whereas kernels in the "Lcals" group are extracted from the Livermore Compiler Analysis Loop Suite [1], and so forth.

As its name suggests, RAJAPerf was initially developed to evaluate the performance of the RAJA framework [3]. To that end, the suite contains multiple versions of each kernel, called variants, that implement the kernel using different programming models. Each kernel is typically implemented in the variants listed below

`Base_Seq`	`Lambda_Seq`	`RAJA_Seq`
`Base_OpenMP`	`Lambda_OpenMP`	`RAJA_OpenMP`
`Base_OMPTarget`		`RAJA_OMPTarget`
`Base_CUDA`	`Lambda_CUDA`	`RAJA_CUDA`
`Base_HIP`	`Lambda_HIP`	`RAJA_HIP`

where `Base_*` are baseline versions of a kernel, implemented using sequential loops, OpenMP, CUDA, etc; `Lambda_*` are variants where the loop bodies (i.e. the kernels *per se*) are defined using lambda functions and called using the corresponding programming model; and `RAJA_*` are variants implemented using RAJA and the corresponding backend. All variants of each kernel perform roughly the same mathematical operations, and the loop body code for each kernel is identical across all variants. This is done to ensure that possible performance differences that may arise across variants are not the result of significantly different implementations (from an algorithmic perspective).

[1] https://github.com/LLNL/RAJAPerf/releases/tag/v0.11.0

Besides timing and other performance metrics, RAJAPerf computes and outputs checksums for all variants run. These checksums are used to ensure that all all variants have run successfully.

2.2 Chapel Performance Suite

As mentioned above, ChapelPerf [4], the Chapel Performance Suite, is a port of RAJAPerf to Chapel. We developed it as a tool to evaluate the performance of Chapel compared to other programming models, and on different computer architectures (namely Arm and x86). We leverage the wide coverage offered by RAJAPerf, both in terms of kernels and variants implemented, to diagnose whether performance issues in Chapel come from Chapel's front-end or back-end compilers, and to compare Chapel with other programming frameworks targeted by RAJAPerf.

Currently, ChapelPerf consists of a mostly complete port of the latest public release of RAJAPerf (v0.11.0). Each kernel in the suite is typically implemented in at least two variants, `Base_Chpl` and `Forall_Chpl`, which correspond to baseline sequential and parallel implementations akin to RAJAPerf's `Base_Seq` and `Base_OpenMP`. In addition, some kernels are also implemented in more idiomatic variants, `Promotion_Chpl` and `Reduction_Chpl`, that make use of Chapel's promotion and reduction constructions to implement certain parallel operations.

Besides implementing RAJAPerf's kernels, ChapelPerf supports the same command-line options as RAJAPerf and outputs the same results in the same formats. This enables ChapelPerf to be run as a drop-in replacement for RAJAPerf, and allows RAJAPerf's checksums to be compared to ChapelPerf's (thereby ensuring the correctness of ChapelPerf's implementations).

3 PORTING EXPERIENCE

Chapel's high-level features and intuitive syntax made porting RAJAPerf a mostly seamless experience. In this section we reflect on this experience, giving emphasis to the main challenges we encountered.

With the few exceptions we mention below, implementing the RAJAPerf kernels in Chapel was a straightforward process. Most kernels were ported directly from their reference C++ implementations by adapting the C++ code into Chapel code, which usually could be achieved by making a small number of changes.

Similarly, porting the logic around the kernels—i.e. the way RAJAPerf is structured, the way the execution is driven, and other similar functionality—was also mostly direct. Most of the mechanisms used in RAJAPerf to implement these features, such as inheritance, polymorphism, the use of various C++ containers, and so on, are either also present in Chapel or have close analogues. Therefore, it was almost always possible to replicate RAJAPerf's approaches in natural and similar-in-spirit implementations in Chapel.

3.1 Challenges

The most challenging kernels to port were kernels that create multiple aliasing "views" of an array (i.e. pointers that alias each other). In C/C++ this is trivial, since we can declare pointers to different regions of an array directly. As an example, consider the kernel `MASS3DPA` that includes multiple declarations like the ones below[2]:

```
double sm1[MDQ * MDQ * MDQ];
double(*DDQ)[MD1][MQ1] = (double(*)[MD1][MQ1])sm1;
double(*QQQ)[MQ1][MQ1] = (double(*)[MQ1][MQ1])sm1;
double(*QDD)[MD1][MD1] = (double(*)[MD1][MD1])sm1;
```

The code above declares `sm1`, a 1D array of doubles, as well as three pointers `DDQ`, `QQQ` and `QDD` that allow accessing it as a 3D array of various dimensions. At present, implementing similar logic in Chapel does not seem to be possible in a straightforward way[3]. In practice, the way these pointers are actually used in RAJAPerf allows us to circumvent the problem and implement the code above by directly aliasing `sm1` with `DDQ`, `QQQ` and `QDD`, i.e.

```
var sm1: [0..<MDQ, 0..<MDQ, 0..<MDQ] real;
ref DDQ = sm1, QQQ = sm1, QDD = sm1;
```

However, we stress that this only works due to the specific way in which aliasing pointers like the ones above are used in RAJAPerf. We also considered how we could implement more general aliasing pointers in Chapel, for cases where that might be needed, and found two main workarounds: one via inline procedures that capture the underlying array data and encapsulate the necessary index arithmetic, e.g.

```
var sm1: [0..<MDQ*MDQ*MDQ] real;
inline proc DDQ(i,j,k) ref return sm1[(i*MD1+j)*MQ1+k];
inline proc QQQ(i,j,k) ref return sm1[(i*MQ1+j)*MQ1+k];
inline proc QDD(i,j,k) ref return sm1[(i*MD1+j)*MD1+k];
```

and another via a wrapper class where the index arithmetic is encapsulated in the class's `this` method[4]. Nevertheless, we believe it would be useful if arrays in Chapel supported this type of aliasing directly, at least in simple one-dimension to multi-dimensions cases.

Another problem we faced was due to the lack of native support for C's `long double` type in Chapel. RAJAPerf makes heavy use of this type to compute checksums of the kernels and variants run, which are used to ensure the correctness of all implementations. Despite not being supported natively, it was relatively easy to work with the `long double` type in Chapel by defining it as an external C type. To that end, we extended the `long double` code from Chapel's LCALS implementation[5] to (i) allow `long double`'s to be mixed with a greater number of native Chapel types, and (ii) allow `long double`'s to be used for input/output directly.

4 PRELIMINARY PERFORMANCE RESULTS

Figure 1 shows early results comparing the performance of the `Base_Chpl` and `Forall_Chpl` variants of ChapelPerf relative to the performance of the `Base_Seq` and `Base_OpenMP` variants of RAJAPerf, respectively. These experiments were run on three distinct systems:

ARCHER2 HPE Cray EX system consisting of 5,860 compute nodes, each with dual AMD EPYC™ 7742 64-core CPUs;

Fulhame HPE Apollo 70 system that consists of 64 compute nodes, each containing two 32-core Marvell ThunderX2 processors;

Isambard 2 HPE Apollo 80 system composed of 72 nodes, each featuring a 48-core A64FX processor.

[2] https://github.com/LLNL/RAJAPerf/blob/v0.11.0/src/apps/MASS3DPA.hpp#L190

[3] https://gitter.im/chapel-lang/chapel?at=618d346838377967f4a74ec9

[4] https://gitter.im/chapel-lang/chapel?at=6196745fabdd6644e390f5b9

[5] https://github.com/chapel-lang/chapel/blob/1.25.0/test/release/examples/benchmarks/lcals/LongDouble.chpl

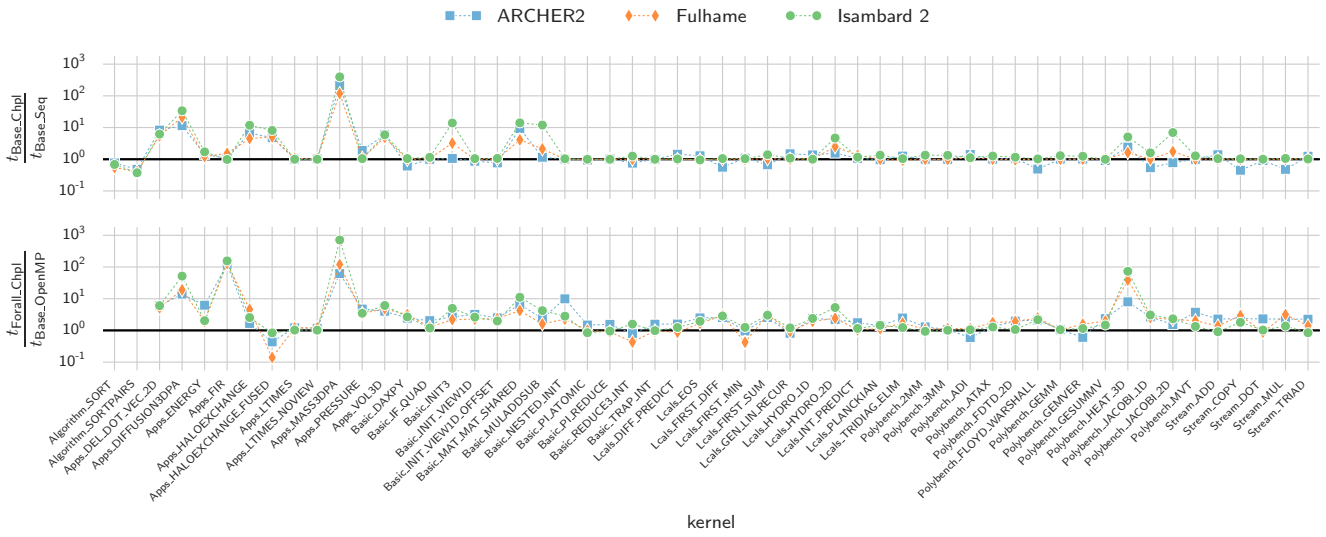

Figure 1: Runtimes for `Base_Chpl` and `Forall_Chpl` variants of ChapelPerf normalised to the runtime of RAJAPerf's `Base_Seq` and `Base_OpenMP` variants (respectively).

The results were obtained with version 1.26 of Chapel, and with the GNU compiler versions 11.2.0 (ARCHER2), 10.3.0 (Isambard 2) and 10.1.0 (Fulhame). The GNU compiler was used both to compile RAJAPerf and as Chapel's back-end compiler, to compile ChapelPerf. ChapelPerf was compiled with `--fast`, whilst RAJAPerf was compiled with `-O3 -mcpu=native` (`-march=native` in the case of ARCHER2). The `Base_Chpl` and `Base_Seq` variants run on a single core of each system, whilst `Forall_Chpl` and `Base_OpenMP` run on all cores of a node.

As the results show, Chapel achieves performance comparable to that of the reference C++ implementations of RAJAPerf on the majority of kernels. The exceptions are a few kernels belonging mainly to the `Apps` group, where, in some cases, Chapel shows significant slowdowns that can range from 10x to 1000x the performance of the corresponding C++ implementations. Additionally, the slowdowns are more pronounced when comparing the parallel variants (`Forall_Chpl` and `Base_OpenMP`), where `Forall_Chpl` usually trails `Base_OpenMP` by at least a few percent. Lastly, the Arm-based systems (Fulhame and Isambard 2) are almost always the ones exhibiting the biggest slowdowns. This suggests that Chapel has not yet been fully optimised for the Arm architecture.

5 CONCLUSIONS AND FUTURE WORK

This work introduces ChapelPerf, a port of the RAJA Performance Suite to Chapel, designed to study the performance of Chapel on different systems and in comparison to other programming languages and runtimes. We discuss the main porting issues we faced, and present preliminary results comparing the performance of Chapel to reference C++ and OpenMP implementations of the RAJAPerf kernels, on three distinct HPE systems. Overall, our results show that the Chapel code is competitive with the reference implementations, except for a relatively small number of kernels. Nevertheless, in a few rare cases, the Chapel code is between 10x and 1000x slower than the reference implementations, which suggests that some code patterns might require more optimisation in Chapel. Finally, on some kernels, the Chapel code runs significantly slower on the Arm-based systems, especially Isambard 2 (A64FX), which suggests there is room for more optimisations that target the Arm architecture.

In the future, we want to finish implementing "idiomatic Chapel" variants of all kernels in ChapelPerf. These variants will enable the usage of ChapelPerf to compare the cost of utilising Chapel high-level features over normal "C-like" loops.

ACKNOWLEDGMENTS

This work used the ARCHER2 UK National Supercomputing Service (https://www.archer2.ac.uk).

The Fulhame HPE Apollo 70 system is supplied to EPCC, the supercomputing centre at the University of Edinburgh, as part of the Catalyst UK programme, a collaboration with Hewlett Packard Enterprise, Arm and SUSE to accelerate the adoption of Arm based supercomputer applications in the UK.

This work used the Isambard 2 UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/T022078/1)

REFERENCES

[1] Richard Hornung. 2013. LCALS: Livermore Compiler Analysis Loop Suite, Version 1.0. https://codesign.llnl.gov/LCALS.php. LLNL-CODE-638939.

[2] Richard Hornung and Holger Hones. 2017. RAJA Performance Suite. Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States). https://doi.org/10.11578/DC.20201001.36

[3] Richard Hornung and Jeffrey Keasler. 2014. *The RAJA Portability Layer: Overview and Status.* Technical Report LLNL-TR–661403, 1169830. Lawrence Livermore National Laboratory (LLNL). https://doi.org/10.2172/1169830

[4] Ricardo Jesus. 2022. ChapelPerf: Chapel Performance Suite. https://github.com/rjjesus/ChapelPerf.