ChapelPerf: A Performance Suite for Chapel

Ricardo Jesus and Michèle Weiland

EPCC, The University of Edinburgh

CHIUW 2022

epcc

THE UNIVERSITY of EDINBURGH

Outline

- Introduction
- Benchmark description
- Porting experience
- Preliminary performance results
- Conclusions and future work

- Application porting and performance optimisation are hard, especially in HPC
  - Plethora of system configurations, variety of codes they are meant to run, etc
- High-level parallel programming languages, libraries, and runtimes, like Chapel, help at coping with this complexity. But
  - How do we know if a given parallel programming framework is good for a specific class of problems/codes?
  - How do we know if it delivers good performance across different system architectures?
  - How do we determine how it compares to other alternatives?

The answer, unsurprisingly, is via benchmarks.

RAJAPerf: The RAJA Performance Suite

- RAJAPerf[1] is a benchmark suite originally developed for RAJA
- It consists of **over 50 loop-based kernels** extracted from HPC applications, other benchmark suites, and similar sources
- Each kernel is **implemented in a number of "variants"** corresponding to different programming models/frameworks

`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`

- The loop body of each kernel is implemented similarly across variants
- A checksum is computed per kernel variant to ensure its correct execution

[1] https://github.com/LLNL/RAJAPerf

ChapelPerf: "The Chapel incarnation of RAJAPerf"

- `https://github.com/rj-jesus/chapelperf`
- **A mostly complete port of RAJAPerf (v0.11.0) to Chapel**
- Developed developed mainly to **evaluate the performance of Chapel** compared to other parallel programming models and across different system architectures
- We have fully implemented two variants of each kernel: `Base_Chpl` and `Forall_Chpl`
  - Working towards implementing more "idiomatic" variants such as `Promotion_Chpl` and `Reduction_Chpl`
- Command-line options and outputs are the same as RAJAPerf's to simplify its usage

- **Overall a straightforward process**
- Most kernels easily ported by adapting the C++ code to Chapel
- Similar experience porting the logic around the kernels (i.e. the way RAJAPerf is structured, how the execution is driven, and so on)
    - Mostly a matter of mapping C++ features such as inheritance, polymorphism, and various containers, to Chapel's analogues

Example with `Apps_FIR` kernel

```
for(RepIndex_type irep = 0; irep < run_reps; ++irep) {
  //#pragma omp parallel for
  for(Index_type i = ibegin; i < iend; ++i ) {
    Real_type sum = 0.0;
    for(Index_type j = 0; j < coefflen; ++j )
      sum += coeff[j]*in[i+j];
    out[i] = sum;
  }
}
```

```
for 0..<run_reps {
  for i in ibegin..<iend {
    var sum: Real_type = 0.0;
    for j in 0..<coefflen do
      sum += coeff[j]*in_[i+j];
    out_[i] = sum;
  }
}
```

```
for 0..<run_reps do
  forall i in ibegin..<iend do
    out_[i] = + reduce (coeff*in_[i..]);
```

Porting challenges: Aliasing array views

- Some RAJAPerf kernels create “aliasing views” over a common array
  - Trivial in C/C++ since we can declare arbitrary pointers to an array directly. Example from `MASS3DPA`[1] to the right

```
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;
```

- Implementing something similar in Chapel in a straightforward manner does not seem to be possible currently. We have found two main workarounds:
  - Utilising inline procedures to capture the underlying array and encapsulate the necessary index arithmetic (right)
  - Using a wrapper class where the index arithmetic is encapsulated in the class’s `this` method[2]

```
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];
```

It would be good if arrays in Chapel supported this type of aliasing natively

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

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

Porting challenges: `long double`

- RAJAPerf uses `long double`'s extensively to compute checksums of runs
- Chapel does not support such a type neither natively nor as a "C type"
- But, relatively easy to work around
  - The implementation of `long double` in LCALS[2] is mostly complete
  - We extended it to **increase interoperability with other Chapel types** and to **enable `long double`'s to be used for input/output**

```
operator :(s: string, type t: longdouble) {
  var ld: longdouble;
  assert(sscanf(s.localize().c_str(), "%Lf",
                c_ptrTo(ld)) == 1);
  return ld;
}
```

```
proc longdouble.writeThis(f) throws {
  var buf = new c_array(c_char, 255);
  var ret = snprintf(buf:c_ptr(c_char), buf.size:size_t, "%Lf",
                     this);
  writer <~> buf:c_string:string;
  return ret;
}
```

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

Preliminary performance results (I)

|epcc|

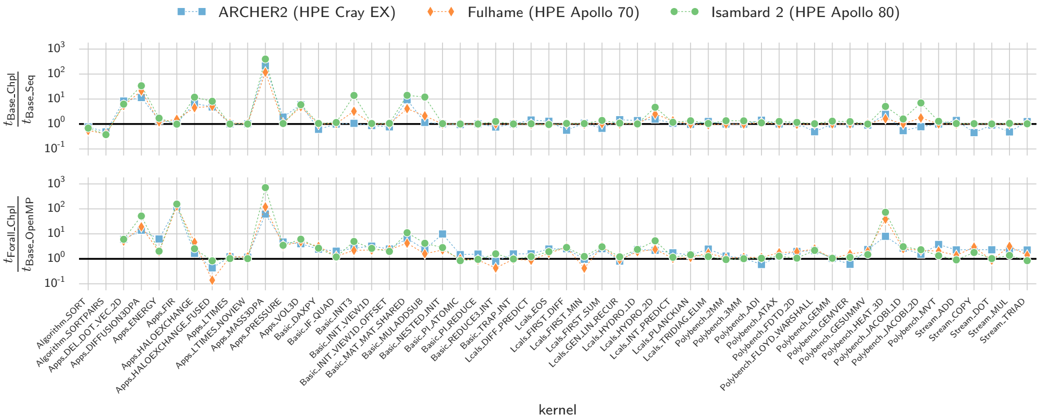
- Comparison between sequential and parallel variants
- Chapel 1.26 using GCC for backend

System	Processor	Compiler	Opt. flags
ARCHER2 (HPE Cray EX)	EPYC 7742	GCC 11.2.0	-O3 -march=native/--fast
Fulhame (HPE Apollo 70)	ThunderX2	GCC 10.1.0	-O3 -mcpu=native/--fast
Isambard 2 (HPE Apollo 80)	A64FX	GCC 10.3.0	-O3 -mcpu=native/--fast

- Most kernels do well compared to the reference C++ sequential/OpenMP versions
- **A few kernels do far worse**
  - Some kernels can run 1000x slower than the reference C++ versions
  - Slowdowns are **more common** when comparing the parallel variants and **more pronounced** on kernels that belong to the `Apps group`
- The Arm-based systems tend to do comparatively worse than the x86 one

(plots on next slide)

Preliminary performance results (II)



Conclusions and future work

|epcc|

- ChapelPerf is an implementation of the RAJAPerf kernels in Chapel
  - Drop-in replacement for RAJAPerf
  - Enables the comparison of Chapel with many other programming models and frameworks across different systems architectures
- Preliminary results show that Chapel overall does well compared to reference implementations, with exceptions
  - Slowdowns can reach 1000x
  - Chapel on Arm-based systems tends to do comparatively worse
  - These results already offer a pointer to code patterns that might necessitate more optimisation on Chapel
- Next steps
  - Identify and address the factors leading to the ocasional reduced performance in Chapel (we are particularly interested in Arm)
  - Implement more variants (idiomatic, GPU-based, multinode?[3])

[3] RAJAPerf recently added preliminary support for this

Questions?

Contact: rjj@ed.ac.uk