Performance Portability of the Chapel Language on Heterogeneous Architectures

Josh Milthorpe
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
Australian National University
Canberra, Australia
ORCID: 0000-0002-3588-9896

Xianghao Wang
Australian National University
Canberra, Australia

Ahmad Azizi
Australian National University
Canberra, Australia

***Abstract*—A performance-portable application can run on a variety of different hardware platforms, achieving an acceptable level of performance without requiring significant rewriting for each platform. Several performance-portable programming models are now suitable for high-performance scientific application development, including OpenMP and Kokkos. Chapel is a parallel programming language that supports the productive development of high-performance scientific applications and has recently added support for GPU architectures through native code generation.**

Using three mini-apps – BabelStream, miniBUDE, and TeaLeaf – we evaluate the Chapel language's performance portability across various CPU and GPU platforms. In our evaluation, we replicate and build on previous studies of performance portability using mini-apps, comparing Chapel against OpenMP, Kokkos, and the vendor programming models CUDA and HIP. We find that Chapel achieves comparable performance portability to OpenMP and Kokkos and identify several implementation issues that limit Chapel's performance portability on certain platforms.

***Index Terms*—performance portability, Chapel language, mini-app, parallel programming, general-purpose GPU programming**

I. Introduction

Chapel is a parallel programming language that supports productive parallel application development on a wide range of computing platforms. Chapel has been used in a range of high-performance scientific applications, including interactive data exploration [1], multi-physics computational fluid dynamics [2], and computational astrophysics [3]. Notably, Chapel supports compilation from a single source file to multiple target architectures, including CPUs and GPUs. This is achieved through the LLVM compiler framework, which allows Chapel code generation to target a diverse range of back-ends, such as PTX for NVIDIA GPUs and AMDGCN for AMD GPUs. Chapel's first-class language features for data parallelism, for example, the **`forall`** parallel loop, are compiled to GPU kernels, with associated host-side code for memory management, kernel launch, and synchronization [4]. While some Chapel language features are not currently supported for GPU, the breadth and quality of support have increased rapidly in recent releases [5]. It is now appropriate to compare Chapel with other heterogeneous programming models that allow single-source programming for diverse hardware platforms.

We seek to answer the question: how well does Chapel support the development of *performance-portable* application codes compared to more widely-used programming models like OpenMP and Kokkos? Performance portability reflects the notion that an application code should not only be portable to a range of different hardware platforms (without requiring rewriting for each new platform) but should also achieve an acceptable level of performance on each platform (without platform-specific optimizations). Deakin et al. [6] compared the performance portability of the OpenMP and Kokkos programming models by using them to write parallel implementations of a set of five mini-apps, each of which represents a key computational pattern common to high-performance scientific computing applications. In a later study, Deakin et al. [7] included SYCL, considering the performance portability of these heterogeneous programming models on CPU platforms only. In this study, we build on their work by undertaking a comparison of Chapel, OpenMP, Kokkos, CUDA, and HIP programming models across of diverse set of hardware platforms. Our contributions are as follows:

- We present the first performance portability study of the Chapel language across a diverse set of computing architectures, including CPUs (Intel, AMD, and ARM) and GPUs (NVIDIA and AMD).
- We replicate and update the published results of Deakin et al. [6]–[10] for the BabelStream, miniBUDE, and TeaLeaf mini-apps on a range of CPU and GPU platforms and present new results for modern AMD GPUs.
- We identify several implementation issues that limit the performance portability of Chapel on these platforms.

II. Mini-Applications in Chapel

We used the Chapel language to create new implementations of three mini-apps developed by the University of Bristol's High Performance Computing research group. These mini-apps have been used extensively to compare parallel programming models and already have idiomatic implementations in OpenMP, Kokkos, CUDA, and HIP. The mini-applications differ in the computational patterns used and, accordingly, in the language features used to implement them in Chapel and other programming models.

A. BabelStream

BabelStream [8] is an update of McCalpin's Stream memory bandwidth benchmark [11], comprising the four kernels from the original benchmark plus a dot product reduction and a modified version of the triad kernel following the Nstream benchmark from Intel's Parallel Research Kernels [12]. In this study, we report memory bandwidth for the stream triad kernel, which computes $A = B + \alpha * C$ for arrays A, B, C. In the CUDA and HIP implementations of BabelStream, each triad of corresponding elements is processed by a single thread, while in the OpenMP and Kokkos versions, data-parallel loops are used (`#pragma omp parallel for` and `Kokkos::parallel_for`). In the Chapel implementation of BabelStream[1], we use a data-parallel `forall` loop as follows:

```
forall i in vectorDom do
  A[i] = B[i] + scalar * C[i];
```

On CPU platforms, this loop is decomposed into multiple chunks to be processed by Chapel worker threads; on the GPU, it is compiled to PTX (NVIDIA) or GCN (AMD) instructions for each GPU thread to process a triad of elements. Chapel also generates the host-side code necessary to launch and synchronize device kernels. Note that the Chapel, OpenMP, and Kokkos versions of BabelStream work for arrays of any size, whereas the CUDA and HIP codes are restricted to array sizes that are an exact multiple of the GPU thread block size. Because of this restriction, the CUDA and HIP kernels avoid checking thread ID against the array upper bound, which may result in a higher achieved bandwidth.

In initial testing with Chapel, we noticed a marked reduction in bandwidth of up to 60% on AMD GPUs due to the cost of kernel launch. Chapel allows launching multiple GPU kernels in parallel from separate asynchronous tasks, which the current Chapel implementation supports by default by creating a separate GPU stream per task. This creates unnecessary overhead for applications like those in the current study, which launch only a single GPU kernel at a time. There was also a small, but statistically significant reduction in bandwidth of less than 2% on NVIDIA GPUs, where the cost of creating separate GPU streams appears to be significantly less. However, the Chapel runtime also supports a single-stream mode in which all GPU kernels are launched on the default stream; we used this mode for all performance results reported in Section III.

B. miniBUDE

miniBUDE [10] is a mini-app created from BUDE [13], a protein docking simulator for drug discovery developed at Bristol University. The miniBUDE kernel computes the energy calculated from each ligand-protein pair and each different position and rotation (i.e. pose) of the ligand. The mini-app is highly arithmetically intensive, making significant use of single-precision arithmetic and trigonometric functions. The main computational kernel is a triple loop nest over proteins, ligands, and poses, which provides ample opportunities for data parallelism that may be decomposed in different ways to suit different target architectures. Our Chapel implementation of miniBUDE[2] closely follows the CUDA implementation in using a one-dimensional kernel and assigning each GPU thread a number of poses, as this exposes the most available parallelism along a single dimension. Like BabelStream, the Chapel miniBUDE GPU kernel is generated from a single data-parallel loop. For accelerator devices, miniBUDE requires data transfer of protein, ligand, and pose information from host to device and energy results from device to host. This is accomplished in Chapel by declaring arrays on the host locale (`here`) and the device locale (`here.gpus[deviceId]`) and copying between them using a simple assignment operator. Chapel automatically generates the host-side code necessary to perform the transfers.

C. TeaLeaf

TeaLeaf [14] is a mini-app that is part of the Mantevo suite [15]. TeaLeaf consists of a number of iterative sparse linear solvers, simulating heat conduction over time using five-point stencils within a two-dimensional grid. Each solver in TeaLeaf exposes grid-based data parallelism within its kernels, similar to BabelStream. Within these parallel stencil kernels, TeaLeaf requires only a small number of arithmetic operations for each load or store, thus, it may be expected to perform better on platforms with higher memory performance, measured in terms of STREAM balance (see Table I).

One of the primary features employed in the Chapel implementation of TeaLeaf[3] is the use of two-dimensional index domains. Compared to other programming models e.g. OpenMP, the application programmer can write solver code in terms of the 'natural' two-dimensional grid using Chapel **`domain`** types, leaving the Chapel compiler to deal with the complexity of converting array indices into memory locations. For example, the `cg_calc_p` kernel of TeaLeaf's conjugate gradient solver is computed over a reduced inner domain within the full grid. This kernel is written for OpenMP target offload as a nested loop, where parallelism is exposed over both loops using the `collapse` directive. The offset `index` into the (one-dimensional) array is calculated within the body of the loop for each two-dimensional grid index (jj, kk) as follows:

```
#pragma omp target teams distribute \
 parallel for simd collapse(2)
for (int jj=halo_depth; jj < y-halo_depth; ++jj) {
  for (int kk=halo_depth; kk < x-halo_depth; ++kk) {
    const int index = kk + jj * x;
    p[index] = beta * p[index] + r[index];
  }
}
```

[1]https://github.com/milthorpe/BabelStream

[2]https://github.com/milthorpe/miniBUDE

[3]https://github.com/milthorpe/TeaLeaf

TABLE I: Processor Configurations and System Balance

Processor	Sockets	Cores	Clock Speed GHz	FP64 TFLOP/s	Memory Bandwidth GB/s	STREAM Balance
Intel Skylake	2	8	3.7	1.89	256	59.2
Intel Cascade Lake	2	24	4.0	6.14	287.3	171.1
Intel Sapphire Rapids	2	52	3.8	12.65	614.4	164.7
AMD Rome	2	64	3.0	6.14	409.6	120
AMD Milan	2	32	3.68	3.77	409.6	73.6
ARM ThunderX2	2	28	2.2	0.99	341.2	23.1
IBM POWER9	2	21	3.5	1.18	340	27.8
NVIDIA P100	2	56	1.19	4.76	549.1	69.4
NVIDIA V100	1	80	1.3	7.83	897	69.9
NVIDIA A100	1	108	1.07	9.75	1935	40.3
AMD MI60	1	64	1.2	7.37	1024	57.6
AMD MI100	1	120	1.0	11.54	1229	75.1

The Kokkos code adopts the reverse approach, decomposing a flattened one-dimensional `index` domain of length $x*y$, and then reconstructing the two-dimensional grid index to perform bounds checks for each element:

```
Kokkos::parallel_for(
  x * y, KOKKOS_LAMBDA(const int &index) {
    const int kk = index % x;
    const int jj = index / x;

    if (kk >= halo_depth
    && kk < x - halo_depth
    && jj >= halo_depth
    && jj < y - halo_depth) {
      p(index) = beta * p(index) + r(index);
    }
  });
```

In contrast to both, the Chapel code performs parallel iteration over a two-dimensional domain and also accesses the array using two-dimensional index tuples `ij`:

```
forall ij in Domain.expand(-halo_depth) {
  p[ij] = beta * p[ij] + r[ij];
}
```

The definition of data domains is independent of the choice of parallel decomposition (distribution), allowing a Chapel developer to easily modify the code to implement and test different decompositions for optimal parallelism.

TeaLeaf makes heavy use of sum reductions to compute global change or error metrics between computations. In Kokkos and OpenMP, these are implemented using language support for reductions (`#pragma omp reduction` and `Kokkos::parallel_reduce`). Chapel also provides language support for reductions (the **`reduce`** intent), however, this is yet to be implemented for GPU code generation. As an interim measure, standalone functions are provided for basic reductions over global memory, such as `gpuSumReduce`, which we use for our implementation of TeaLeaf. To use these functions requires that the partial results of the kernel be stored in an array in GPU global memory. For example, in the `cg_calc_ur` kernel of TeaLeaf, the partial results for each thread r_i^2 are stored in the array `temp`, which is then reduced to compute the sum `rrn`.

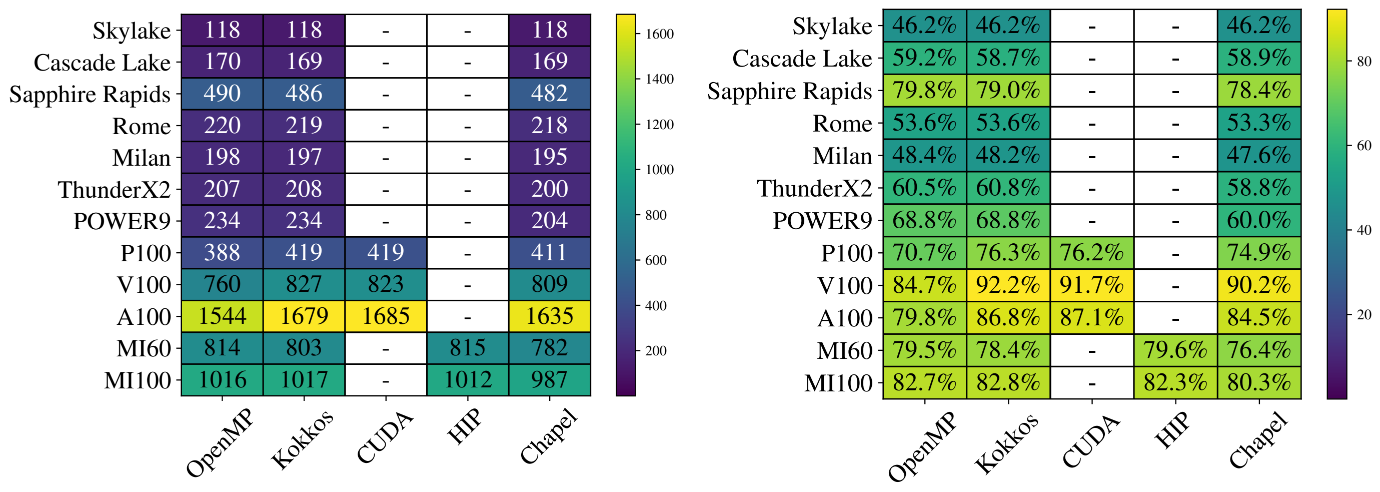
```
var temp: [reduced_local_domain] real = noinit;
...
forall ij in reduced_local_domain {
  u[ij] += alpha * p[ij];
  r[ij] -= alpha * w[ij];
  temp[ij] = r[ij] ** 2;
}
var rrn = gpuSumReduce(temp);
```

We expect these standalone functions may be replaced in future by code generation using the LLVM backends for each GPU device, which could take advantage of fast shared memory to compute partial reductions inside a GPU thread block. With this change, the GPU code would become identical to the CPU code, as follows:

```
var rrn: real;
forall ij in reduced_local_domain
  with (+ reduce rrn) {
  u[ij] += alpha * p[ij];
  r[ij] -= alpha * w[ij];
  rrn += r[ij] ** 2;
}
```

Thus, by eliminating the need for a temporary array, the performance of Chapel reductions may improve in the future.

Unlike the other programming models evaluated in this study, Chapel provides native language support for multi-dimensional index domains. In TeaLeaf, this allows array data to be defined and accessed using the natural two dimensions (x,y) over which the physical problem is defined, rather than by iterating over a single dimension and including index calculations (integer modulo and division) inside each loop. Using two-dimensional indices improved the readability of the Chapel code and performed well on CPU platforms. However, in preliminary testing, we found that using two-dimensional domains reduced GPU performance due to the under-utilization of the available GPU cores. In Chapel 1.33, when generating a GPU kernel for a multi-dimensional domain, the first dimension of the domain is assigned to individual GPU threads, with the remaining dimensions implemented as loops inside the GPU kernel. This means that the available GPU thread parallelism is limited by a single dimension of the domain rather than (as with the other GPU programming



(a) Memory bandwidth in GB/s, higher is better

(b) Architectural efficiency, higher is better

Fig. 1: BabelStream Triad results for arrays of length 2^{28} FP64 elements

models) the product of all dimensions. To work around this issue, at the suggestion of the Chapel development team, we replaced multi-dimensional loops with a one-dimensional iteration over a linearized index space. For example, the following two-dimensional loop:

```
const Domain = {0..<y, 0..<x};
foreach ij in Domain {
  u[ij] = energy[ij] * density[ij];
}
```

would be replaced in our modified Chapel code with:

```
const Domain = {0..<y, 0..<x};
const OneD = {0..<y*x};
foreach oneDIdx in OneD {
  const ij = local_domain.orderToIndex(oneDIdx);
  u[ij] = energy[ij] * density[ij];
}
```

This workaround allows full utilization of the GPU when working with multi-dimensional arrays. We believe it should be possible to implement a compiler transformation to apply this linearization for common GPU loops or, better still, to faithfully translate two- and three-dimensional loops to equivalent two- and three-dimensional PTX or GCN kernels. In Section III, we report results for TeaLeaf both with and without this workaround.

III. Experimental Evaluation

A. Experimental Platforms

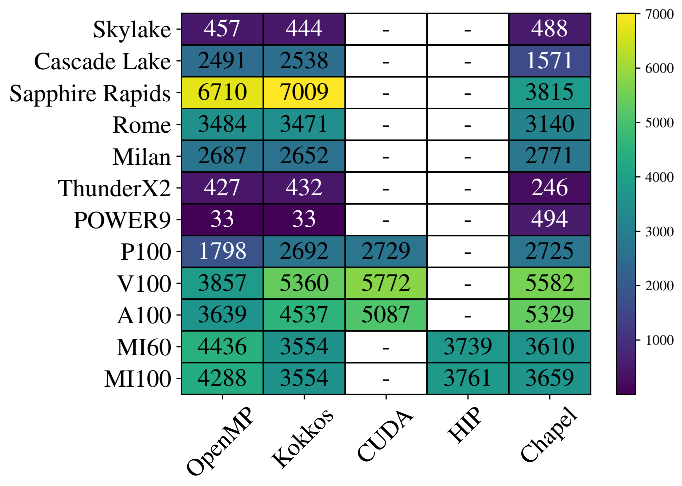
We ran the Chapel, OpenMP, and Kokkos implementations of the mini-apps on twelve varied hardware platforms from multiple processor generations, including Intel, AMD, ARM, and POWER CPUs, NVIDIA GPUs, and AMD GPUs. We also ran the CUDA implementation on NVIDIA GPUs and the HIP implementation on AMD GPUs, as we expected these 'native' GPU programming models to provide a gold standard for performance on their respective platforms. Table I gives details of all processors (both CPU and GPU) we used in our evaluation, gathered from vendor-provided architectural specifications. For each platform, we report the architectural 'STREAM' balance [11] defined as $\mathrm{GFLOP\,s^{-1}}/\mathrm{Gword\,s^{-1}}$ i.e. the number of 64-bit floating point operations that can be completed per 64-bit word loaded from memory.

We used Chapel 1.33 for all platforms[4]; when running on GPUs, we set `--gpuUseStreamPerTask=false` to avoid creating a separate stream for each GPU kernel invocation. We used Kokkos version 4.2.0 on all platforms, with the same compiler used as OpenMP for each CPU platform, or `hipcc` or `nvcc` as appropriate for the GPUs. The compiler versions used for OpenMP, CUDA, and HIP on each platform are shown in Table V at the end of this section. All benchmark scripts, including compiler versions and configuration settings for each platform, are available in our GitHub fork of the Bristol HPC Performance Portability Studies framework at https://github.com/milthorpe/performance-portability.

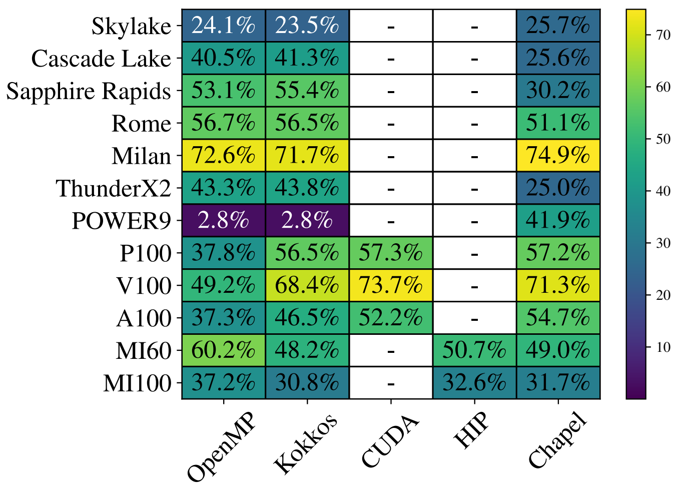
B. Results

1) BabelStream: We ran BabelStream version 5.0 on all platforms using arrays of 2^{28} 64-bit floating-point elements. This means each array is 2 GiB in size and, therefore, four times the size of the largest last-level cache on any of the test platforms, which is the AMD Rome CPU with an L3 cache size of 512 MiB. We compared the maximum memory bandwidth recorded for the stream triad kernel on each platform. Figure 1a shows the memory bandwidth in GB/s, and Figure 1b gives the architectural efficiency in terms of the percentage of peak memory bandwidth for each platform. Where a programming model is not available for a system (e.g. CUDA for AMD GPUs) or would be inappropriate to compare (e.g. HIP for NVIDIA GPUs), no result is recorded.

[4]except for miniBUDE on Cascade Lake, Sapphire Rapids, ThunderX2, and POWER CPUs, for which we used a pre-release version of Chapel 2.0 - see Section III-B2



(a) Effective GFLOP/s, higher is better



(b) Architectural efficiency, higher is better

Fig. 2: miniBUDE results for small deck bm1

The NVIDIA A100 GPU achieved the highest absolute bandwidth for all programming models, which was expected given this system's impressive specified memory bandwidth of 1935 GB/s. The highest architectural efficiencies (percentage of peak bandwidth achieved) were recorded on the NVIDIA V100 GPU. The vendor-specific programming models (CUDA/HIP) achieved the highest efficiencies on their respective platforms, except for the NVIDIA V100 and A100, where Kokkos outperformed CUDA.

TABLE II: Performance portability metric for BabelStream

	Φ by programming model (architectural efficiency %)				
Platform set	OpenMP	Kokkos	CUDA	HIP	Chapel
All platforms	65.0	65.8	0	0	64.4
Supported CPUs	58.5	58.3	0	0	57.1
Supported GPUs	79.2	82.9	84.5	80.9	80.9

Pennycook, Sewall, and Lee [16] define a performance portability metric Φ as the harmonic mean of the efficiencies on each platform. Table II shows the metric Φ for each programming model as calculated from the architectural efficiencies shown in Figure 1b. Of the three portable programming models, Kokkos achieved the highest Φ of 65.8; OpenMP was second with 65.0, and Chapel was third with 64.4. By definition, Φ is zero for any programming model that is not supported on one or more platforms in the set. Furthermore, combining performance efficiencies across CPUs and GPUs can obscure important differences between programming models. For these reasons, Table II also shows Φ calculated for each model across the set of CPUs and GPUs supported by that model. CUDA and HIP achieve good performance portability across the subsets of GPUs they support. It is also apparent that the lower value of Φ for OpenMP is entirely due to poorer results on GPU platforms, whereas on CPUs, OpenMP achieves the best performance portability of any of the tested models. Deakin et al. [7] evaluated different programming models and compilers for various CPU platforms and found that OpenMP achieved the highest performance portability; our results accord with their findings.

2) miniBUDE: We ran miniBUDE version 2.0 on all platforms using the small 'bm1' input deck. We compared the application-reported GFLOP/s measure on each platform. Figure 2a shows the effective raw performance in GFLOP/s, and Figure 2b gives the architectural efficiency in terms of the percentage of peak memory bandwidth for each platform.

The initial performance of Chapel on Cascade Lake, Sapphire Rapids, ThunderX2, and POWER CPUs was extremely poor (architectural efficiencies of 7%, 3%, 12%, and 13% respectively). This is due to an older version of glibc (2.28) provided by the operating systems on these platforms.[5] Chapel 1.33 implements the `sqrt` and `abs` functions for 32-bit floating point operands as calls to the glibc functions `sqrt` and `fabsf`, which in older versions of glibc do not use the available fast vector instructions available on Intel architectures. This issue has been resolved in Chapel 2.0 (prerelease), so for those platforms using the older glibc, we built miniBUDE using a pre-release version of Chapel.

For miniBUDE, the Sapphire Rapids CPU achieved the highest absolute performance of 7009 GFLOP/s with Kokkos, reflecting this CPU's superior floating point arithmetic capabilities. The NVIDIA V100 GPU achieved the highest performance among GPUs at 5772 GFLOP/s with CUDA, and slightly lower with Chapel and Kokkos. This was surprising given the A100 nominally has a higher peak FLOPs rate at 9.75 TFLOP/s compared to 7.83 TFLOP/s for the V100. We note that the V100 DGX has a higher boost clock rate, and we speculate that this application may not have benefited from the nominal increase in FLOPs per cycle that the A100 provides.

[5]https://github.com/chapel-lang/chapel/issues/24112

The AMD CPUs recorded the highest architectural efficiencies, in terms of the percentage of peak FLOP/s achieved. OpenMP, Kokkos, and Chapel all achieved greater than 70% efficiency on both Rome and Milan CPU platforms. Architectural efficiencies were poor on the other CPU platforms; in particular, both OpenMP and Kokkos achieved only 2.8% of peak FLOP/s on POWER9. CUDA again achieved the highest architectural efficiencies on NVIDIA GPUs, however, OpenMP was the best-performing model on AMD GPUs.

TABLE III: Performance portability metric for miniBUDE

	Φ by programming model (architectural efficiency %)				
Platform set	OpenMP	Kokkos	CUDA	HIP	Chapel
All platforms[a]	42.9	44.6	0	0	38.5
Supported CPUs[a]	43.0	43.1	0	0	32.6
Supported GPUs	42.7	46.7	59.8	39.7	49.1

[a]Except POWER9.

Table III shows the metric Φ for each programming model as calculated from the architectural efficiencies shown in Figure 2b, as well as Φ calculated for each model across the set of CPUs and GPUs supported by that model. Because the architectural efficiency on POWER9 is so low for OpenMP and Kokkos, it skews the Φ metric wildly. Thus, we chose to exclude POWER9 in calculating performance portability for miniBUDE. Of the three portable programming models, Kokkos again achieved the highest Φ of 44.6; OpenMP was second with 42.9, and Chapel was third with 38.5. Kokkos and OpenMP performed comparably on CPUs, while Chapel achieved the highest performance portability across GPUs.

On Intel CPU platforms, we noted that an unintended consequence of a feature of Chapel's LLVM code generation caused roughly a 50% drop in performance. Before code generation, the Chapel compiler normalizes conditional expressions by converting them into conditional statements.[6] Unfortunately, this can hinder LLVM's standard constant-folding optimizations, which, for miniBUDE on Intel, resulted in the generation of conditional instructions rather than constant register values. For example, the miniBUDE `fasten_main` loop contains the following code:

```
param NPNPDIST = 5.5;
param NPPDIST = 1.0;
const distdslv =
  if phphb_ltz
  then (
    if lhphb_ltz
    then NPNPDIST
    else NPPDIST
  ) else (
    if lhphb_ltz
    then NPPDIST
    else -max(real(32))
);
const r_distdslv = 1.0 / distdslv;
```

[6]https://github.com/chapel-lang/chapel/issues/21229

Each of the four conditional branches results in the assignment of `distdslv` to a compile-time constant; therefore, the inverse `r_distdslv` is also a compile-time constant. However, for the Intel architecture, the generated machine code computes `r_distdslv` using a slow floating-point division due to the failure of the constant folding optimization. We evaluated the effect of constant folding by writing a similar conditional expression for `r_distdslv`, which ensured it was computed by choosing between constant values. This improved the performance on Intel Sapphire Rapids to 6.4 TFLOP/s, an architectural efficiency of 50.5% comparable to that achieved by OpenMP and Kokkos on the same platform. This issue did not affect Chapel code generation for the AMD, ARM, or POWER CPU platforms. If this issue were addressed, Chapel's performance portability for miniBUDE would be 46.1% across all platforms.

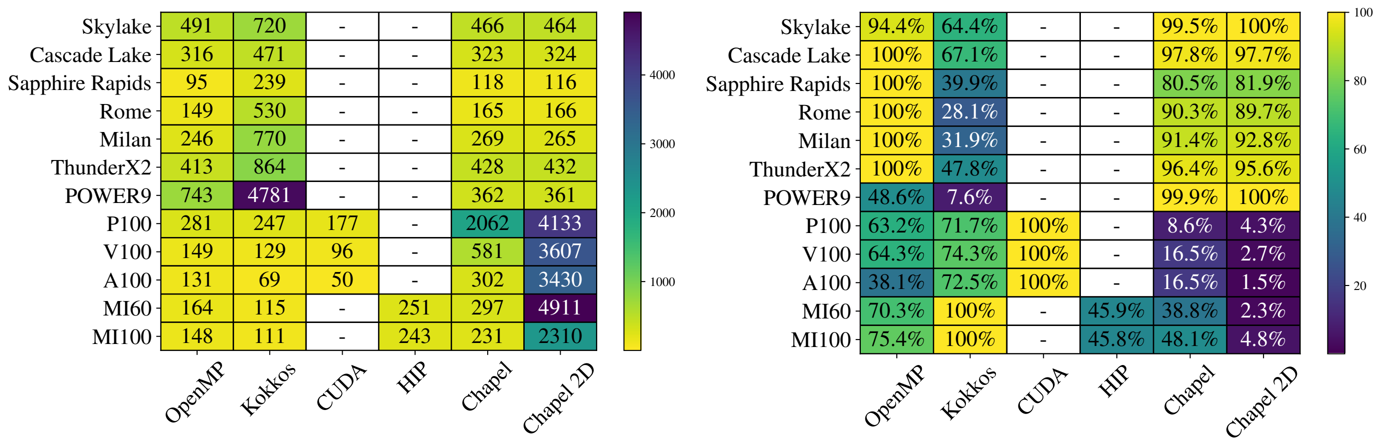
While the Chapel implementation of miniBUDE achieved higher performance on NVIDIA GPUs than either OpenMP or Kokkos, it still fell behind the CUDA implementation. In investigating this issue, we noticed that the Chapel compiler fails to take advantage of register coalescing optimizations provided by LLVM, which are available to the GPU backend code generation. This results in a higher register pressure and, thus, lower warp occupancy, reducing parallel execution. We expect that taking full advantage of existing LLVM optimizations could enable the Chapel implementation to match or even beat CUDA's performance on NVIDIA platforms.

Previous evaluations of miniBUDE [7], [10] required separate OpenMP implementations for CPU and GPU using OpenMP target offload. Poenaru et al. [10] noted in 2021 that such specialization of an application for different platforms runs counter to the notion of performance portability, and favorably noted that the Kokkos implementation of miniBUDE was able to support all platforms with a single code base. However, the OpenMP results reported here are for version 2 of miniBUDE, which includes a unified OpenMP implementation that may be considered truly performance-portable.

3) TeaLeaf: We ran TeaLeaf version 2.0 on all platforms using the `tea_bm_5.in` input configuration, which performs a conjugate gradient solve on a 4000×4000 grid for 10 iterations. As TeaLeaf does not report an architectural efficiency (e.g. GFLOP/s, we use the total runtime (elapsed wall clock time) as our figure of merit.

Figure 3a shows the effective raw performance in GFLOP/s, and Figure 3b gives the application efficiency, which is calculated by dividing the execution time of the highest-performing implementation by the execution time of each model. By this definition, the highest-performing model on each platform has an application efficiency of 100%.

Results for TeaLeaf are more mixed than for the other applications, reflecting its more complex nature. Chapel performs best on Intel and POWER CPUs; OpenMP performs best on AMD and ThunderX2 CPUs; and Kokkos performs best on GPUs (other than NVIDIA GPUs where CUDA performs best). Each of the portable programming models returns at least one very poor result: OpenMP on the A100,



(a) Execution time (s), lower is better

(b) Application efficiency, higher is better

Fig. 3: TeaLeaf results for input `tea_bm_5.in`

Kokkos on AMD CPUs, and Chapel on NVIDIA GPUs. However, OpenMP achieved significantly higher performance portability overall. The 'Chapel' column in each figure gives the results for one-dimensional GPU loops; for comparison, the 'Chapel 2D' column shows the results for the unmodified code using two-dimensional iteration on GPUs. It is clear that the reduction in the number of GPU threads due to Chapel parallelizing over only the first dimension of the domain results in a disastrous loss of performance for TeaLeaf.

TABLE IV: Performance portability metric for TeaLeaf

	Φ by programming model (application efficiency %)				
Platform set	OpenMP	Kokkos	CUDA	HIP	Chapel
All platforms	73.6	37.4	0	0	35.2
Supported CPUs	87.4	27.9	0	0	94.0
Supported GPUs	58.8	81.7	100.0	45.9	17.6

Table IV shows the metric Φ for each programming model as calculated from the architectural efficiencies shown in Figure 3b for all platforms and for CPUs and GPUs supported by each model. OpenMP had the highest performance portability overall; Chapel had the highest performance portability on CPUs; Kokkos performed best of the portable models on GPUs; and CUDA achieved Φ of 100% on supported GPUs as it was the best-performing model on all NVIDIA GPUs.

Deakin et al. [6] found in 2019 that OpenMP had poor application efficiency on GPUs, concluding that this was due to the immaturity of the target offloading implementation in major compilers, including Clang-LLVM and expecting OpenMP portability to improve over time. Our results confirm their expectations: LLVM's target offload implementation has improved markedly up to clang-17.0.6.

IV. Conclusion

Each of the mini-apps discussed in this paper exercises different features of the language implementations that we have considered. When applied over a varied set of applications, we have found the Φ metric of Pennycook, Sewall, and Lee [16] to be a useful lens for evaluating portable programming models and identifying areas of relative strength and weakness. In replicating and updating previous work on performance portability of the OpenMP and Kokkos models [6]–[10], we note that the performance of both OpenMP and Kokkos continues to improve on existing platforms, as portability extends to new platforms. To this work, we have added Chapel as a new portable programming model for comparison and demonstrated that its performance portability is comparable to existing models. However, we have also identified a number of implementation issues that reduce Chapel's performance portability on certain platforms, in some cases requiring code to be significantly modified to perform well on GPUs. If these issues can be fixed, users will be able to develop new applications without experiencing performance pitfalls.

Acknowledgments

This research used resources of the Experimental Computing Laboratory (ExCL) and the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which are supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725, and resources of the National Computational Infrastructure (NCI), which is supported by the Australian Government.

This research was funded, in part, by the Brisbane Advanced Programming Systems Project.

The authors thank Engin Kayraklioglu, Brad Chamberlain, Michael Ferguson, and Jade Abraham from the HPE Chapel team for helpful discussions during our development of the mini-applications in Chapel.

TABLE V: Software Configuration

Processor	Operating System	GPU Driver Version	Compiler	Chapel version
Intel Skylake	Ubuntu 20.04.6		clang 17.0.6	1.33
Intel Cascade Lake	Ubuntu 22.04.3		clang-17.0.1	2.0 pre-release
Intel Sapphire Rapids	Ubuntu 22.04.3		clang-17.0.1	2.0 pre-release
AMD Rome	Ubuntu 22.04.3		clang 17.0.6	1.33
AMD Milan	Ubuntu 22.04.3		clang 17.0.6	1.33
ARM ThunderX2	CentOS Stream 8		clang 17.0.2	2.0 pre-release
IBM POWER9	CentOS 8.3		gcc 10.2	2.0 pre-release
NVIDIA P100	Ubuntu 20.04.6	525.147.05	nvcc 11.5	1.33
NVIDIA V100	Ubuntu 22.04.3	535.129.03	nvcc 12.3	1.33
NVIDIA A100	Ubuntu 22.04.3	545.23.08	nvcc 12.3	1.33
AMD MI60	Ubuntu 22.04.3	5.18.13	hipcc 5.4.3	1.33
AMD MI100	Ubuntu 22.04.3	5.18.13	hipcc 5.4.3	1.33

References

[1] M. Merrill, W. Reus, and T. Neumann, "Arkouda: interactive data exploration backed by Chapel," in *6th Annual Chapel Implementers and Users Workshop*, 2019.

[2] M. Parenteau, S. Bourgault-Cote, F. Plante, E. Kayraklioglu, and E. Laurendeau, "Development of parallel CFD applications with the Chapel programming language," in *AIAA Scitech 2021 Forum*, 2021, p. 0749.

[3] N. Padmanabhan, E. Ronaghan, J. L. Zagorac, and R. Easther, "Simulating ultralight dark matter in Chapel," in *7th Annual Chapel Implementers and Users Workshop*, 2020.

[4] E. Kayraklioglu, A. Stone, D. Iten, S. Nguyen, M. Ferguson, and M. Strout, "Targeting GPUs using Chapel's locality and parallelism features," in *9th Annual Chapel Implementers and Users Workshop (CHIUW)*, 2022. [Online]. Available: https://chapel-lang.org/CHIUW/2022/Kayraklioglu.pdf

[5] E. Kayraklioglu, A. Stone, and D. Fedorin, "Recent GPU programming improvements in Chapel," in *10th Annual Chapel Implementers and Users Workshop (CHIUW)*, 2023. [Online]. Available: https://chapel-lang.org/CHIUW/2023/Kayraklioglu.pdf

[6] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, and J. Salmon, "Performance portability across diverse computer architectures," in *IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2019.

[7] T. Deakin, J. Cownie, W.-C. Lin, and S. McIntosh-Smith, "Heterogeneous programming for the homogeneous majority," in *IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2022.

[8] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Evaluating attainable memory bandwidth of parallel programming models via BabelStream," *International Journal of Computational Science and Engineering*, vol. 17, no. 3, pp. 247–262, 2018.

[9] T. Deakin, A. Poenaru, T. Lin, and S. McIntosh-Smith, "Tracking performance portability on the yellow brick road to exascale," in *IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2020.

[10] A. Poenaru, W.-C. Lin, and S. McIntosh-Smith, "A performance analysis of modern parallel programming models using a compute-bound application," in *International Conference on High Performance Computing*, 2021.

[11] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," in *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, Dec 1995, p. 19–25.

[12] R. F. Van der Wijngaart and T. G. Mattson, "The parallel research kernels," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2014.

[13] S. McIntosh-Smith, J. Price, R. B. Sessions, and A. A. Ibarra, "High performance in silico virtual drug screening on many-core processors," *International Journal of High Performance Computing Applications*, vol. 29, no. 2, pp. 119–134, 2015.

[14] S. McIntosh-Smith, M. Martineau, T. Deakin, G. Pawelczak, W. Gaudin, P. Garrett, W. Liu, R. Smedley-Stevenson, and D. Beckingsale, "Tealeaf: A mini-application to enable design-space explorations for iterative sparse linear solvers," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2017.

[15] P. S. Crozier, H. K. Thornquist, R. W. Numrich, A. B. Williams, H. C. Edwards, E. R. Keiter, M. Rajan, J. M. Willenbring, D. W. Doerfler, and M. A. Heroux, "Improving performance via mini-applications," 2009.

[16] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, vol. 92, pp. 947–958, 2019.