OAK RIDGE National Laboratory

Performance Portability of the Chapel Language on Heterogeneous Architectures

Heterogeneity in Computing Workshop
IPDPS, 27 May 2024

Josh Milthorpe,
Xianghao Wang, and Ahmad Azizi

ORNL Advanced Computing Systems Research, milthorpejj@ornl.gov
Australian National University

ORNL is managed by UT-Battelle LLC for the US Department of Energy

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

U.S. DEPARTMENT OF ENERGY

Chapel Programming Language

- Parallel programming language supporting productive app development, including:
  - data exploration
  - multi-physics CFD
  - computational astrophysics
- Single-source compilation to multiple targets through LLVM
- First-class language features for task & data parallelism, synchronization, distributed memory
- Rapidly-improving GPU support
  - host-side code gen for memory management, kernel launch, synchronization
  - NVIDIA (LLVM PTX backend)
  - AMD (GCN backend)

Performance Portability

- Application code should run on many different hardware platforms …
  - (without requiring rewriting for each new platform)
- … and achieve acceptable performance on each platform
  - (without platform-specific optimizations)
- Pennycook, Sewall, and Lee's metric Φ : harmonic mean of efficiency on each platform
  - Architectural efficiency e.g. fraction of peak FLOP/s
  - Application efficiency e.g. inverse speedup versus fastest version
  - Φ = 0 if code doesn't run on all platforms
- How well does Chapel support development of performance-portable application codes compared to more widely-used programming models like OpenMP and Kokkos?

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.

Mini-apps

- We created new Chapel implementations of three mini-apps developed by the University of Bristol's High Performance Computing group
- These miniapps have been used extensively to compare parallel programming models and already have idiomatic implementations in OpenMP, Kokkos, CUDA, and HIP.
  - BabelStream: streaming memory access
  - miniBUDE: numerically intensive molecular dynamics
  - TeaLeaf: memory-intensive stencil PDE solver
- Not included in this study:
  - multi-device
  - distributed memory
  - programmer productivity

BabelStream

- An update of McCalpin's Stream memory bandwidth benchmark, comprising:

Kernel	Function	Load/Store	FLOP
Copy	C = A	2	0
Add	C = A + B	3	1
Mul	B = α * C	2	1
Triad	A = B + α * C	3	2 (1 FMA)
Nstream (PRK)	A += B + α * C	4	3 (1 FMA)
Dot	x = A . B	2	2 (1 FMA)

- We measure BabelStream version 5.0 triad with 2^{28} 64-bit FP elements

https://github.com/milthorpe/BabelStream

Deakin, T., Price, J., Martineau, M., & McIntosh-Smith, S. (2018). Evaluating attainable memory bandwidth of parallel programming models via BabelStream. International Journal of Computational Science and Engineering, 17(3), 247-262.

BabelStream Triad Implementations

- Chapel

```
proc triad() {
  forall i in vectorDom do
    A[i] = B[i] + scalar * C[i];
  }
}
```

```
const streamLocale = if useGPU
    then here.gpus[deviceIndex]
    else here;
on streamLocale do {
  const vectorDom = 0..#arraySize;
  var A, B, C: [vectorDom] eltType = noinit;
}
```

  - CPU: loop is decomposed into chunks to be executed by worker threads
  - GPU: compiled to PTX (NVIDIA) or GCN (AMD) for each threads to compute a triad of elements; compiler generates host-side code for kernel launch and synchronization

- CUDA

```
template <class T>
void CUDAStream<T>::triad()
{
  triad_kernel<<<array_size/TBSIZE, TBSIZE>>>
    (d_a, d_b, d_c);
  check_error();
  cudaDeviceSynchronize();
  check_error();
}
```

```
template <typename T>
__global__ void triad_kernel(T * a, const T * b,
const T * c)
{
  const T scalar = startScalar;
  const int i = blockDim.x * blockIdx.x +
threadIdx.x;
  a[i] = b[i] + scalar * c[i];
}
```

BabelStream Triad Implementations (2)

- Kokkos

```
template <class T>
void KokkosStream<T>::triad()
{
  Kokkos::View<T*> a(*d_a);
  Kokkos::View<T*> b(*d_b);
  Kokkos::View<T*> c(*d_c);

  const T scalar = startScalar;
  Kokkos::parallel_for(array_size, KOKKOS_LAMBDA (const long index)
  {
    a[index] = b[index] + scalar*c[index];
  });
  Kokkos::fence();
}
```

BabelStream Triad Implementations (3)

- OpenMP

```
template <class T>
void OMPStream<T>::triad()
{
  const T scalar = startScalar;

#ifdef OMP_TARGET_GPU
  int array_size = this->array_size;
  T *a = this->a;
  T *b = this->b;
  T *c = this->c;
  #pragma omp target teams distribute parallel for simd
#else
  #pragma omp parallel for
#endif
  for (int i = 0; i < array_size; i++)
  {
    a[i] = b[i] + scalar * c[i];
  }
  #if defined(OMP_TARGET_GPU) && defined(_CRAYC)
  // If using the Cray compiler, the kernels do not block, so this update forces
  // a small copy to ensure blocking so that timing is correct
  #pragma omp target update from(a[0:0])
  #endif
}
```

Experimental Platforms

	Processor	Sockets	Cores	Clock GHz	FP TFLOP/s	Mem BW GB/s	STREAM Balance*
CPU	Intel Skylake	2	8	3.70	1.89	256.0	59.2
	Intel Cascade Lake	2	24	4.00	6.14	287.3	171.1
	Intel Sapphire Rapids	2	52	3.80	12.65	614.4	164.7
	AMD Rome	2	64	3.00	6.14	409.6	120.0
	AMD Milan	2	32	3.68	3.77	409.6	73.6
	ARM ThunderX2	2	28	2.20	0.99	341.2	23.1
	IBM POWER9	2	21	3.50	1.18	340.0	27.8
GPU	NVIDIA P100	1	56	1.19	4.76	549.1	69.4
	NVIDIA V100	1	80	1.30	7.83	897.0	69.9
	NVIDIA A100	1	108	1.07	9.75	1935.0	40.3
	AMD MI60	1	64	1.20	7.37	1024.0	57.6
	AMD MI100	1	120	1.00	11.54	1229.0	75.1
	AMD MI250X	1	110	1.00	23.94	1600.0	119.7

* GFLOP s^{-1} / Gword s^{-1}

Experimental Configuration

	Processor	Operating System	GPU Driver Version	Compiler
CPU	Intel Skylake	Ubuntu 20.04.6		clang 17.0.6
	Intel Cascade Lake	Ubuntu 22.04.3		clang 17.0.1
	Intel Sapphire Rapids	Ubuntu 22.04.3		clang 17.0.1
	AMD Rome	Ubuntu 22.04.3		clang 17.0.6
	AMD Milan	Ubuntu 22.04.3		clang 17.0.6
	ARM ThunderX2	CentOS Stream 8		clang 17.0.2
	IBM POWER9	CentOS 8.3		gcc 10.2
GPU	NVIDIA P100	Ubuntu 20.04.6	525.147.05	nvcc 11.5
	NVIDIA V100	Ubuntu 22.04.3	550.54.15	nvcc 12.3
	NVIDIA A100	Ubuntu 22.04.3	555.42.02	nvcc 12.3
	AMD MI60	Ubuntu 22.04.3	6.3.6	hipcc 5.4.3
	AMD MI100	Ubuntu 22.04.3	5.15.0-15	hipcc 5.4.3
	AMD MI250X	SUSE LES 15.4	6.3.6	hipcc 5.4.3

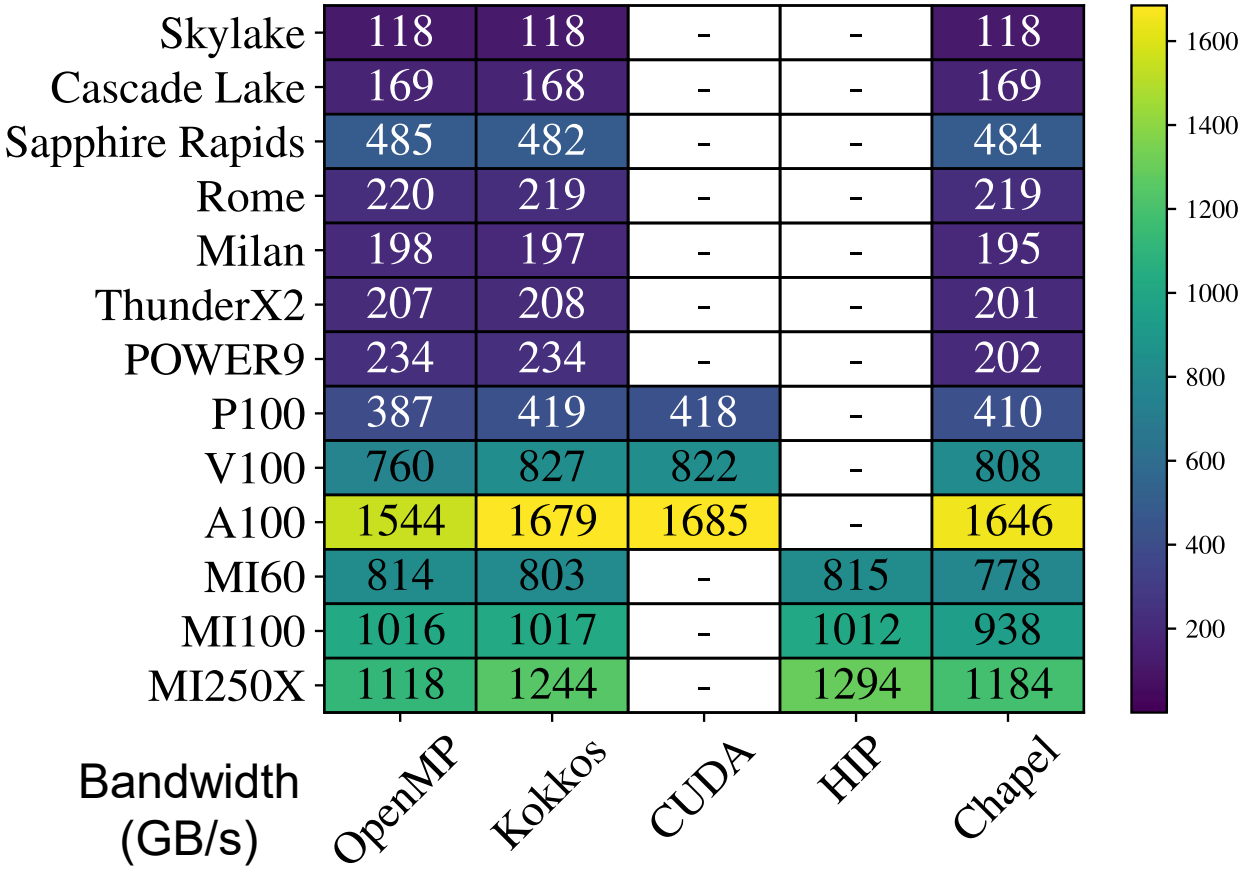
Chapel 2.0, Kokkos 4.2.0

https://github.com/milthorpe/performance-portability

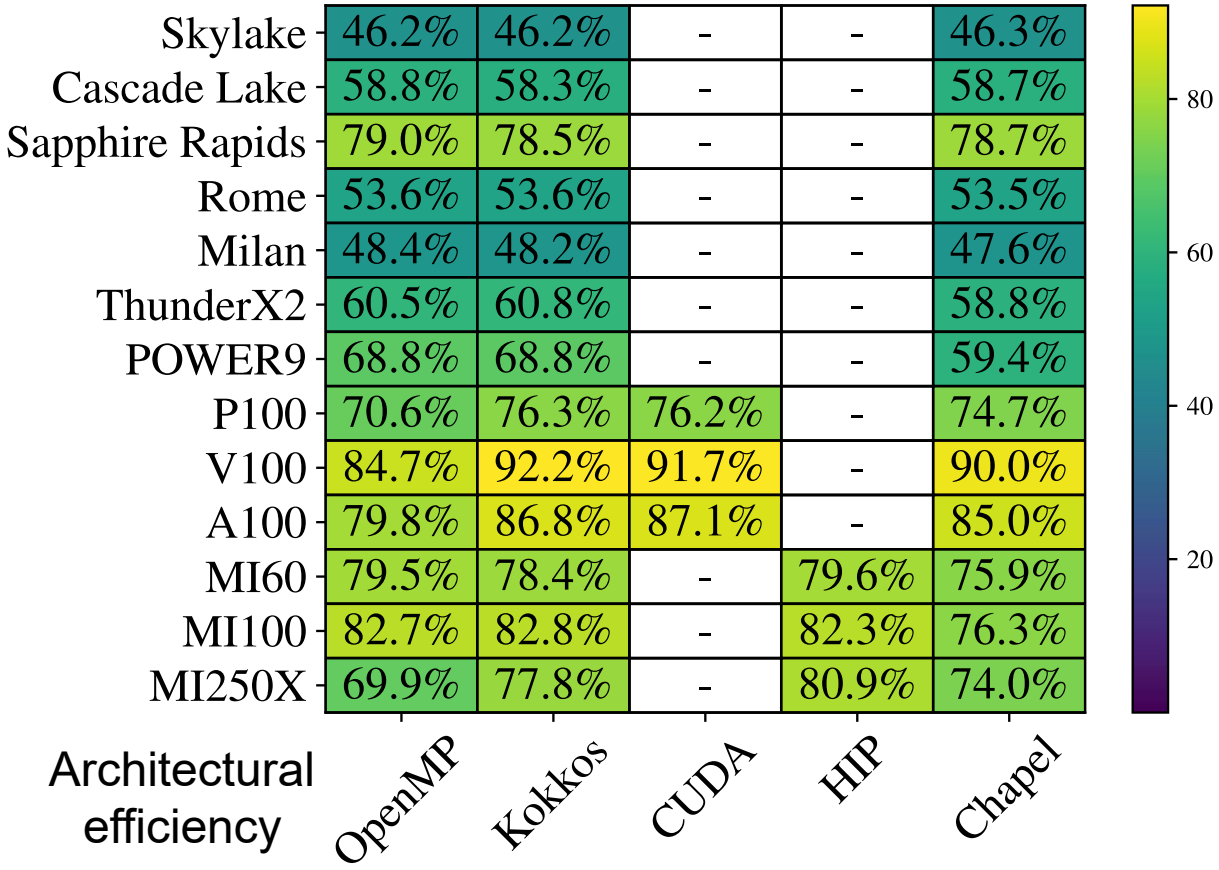
BabelStream Performance Portability

BabelStream v5 triad – 2^{28} 64-bit elements

Bandwidth (GB/s)



Architectural efficiency



Platforms	OpenMP	Kokkos	CUDA	HIP	Chapel
All platforms	64.9%	**65.8%**	0	0	64.0%
Supported CPUs	**57.5%**	57.4%	0	0	56.1%
Supported GPUs	79.1%	**82.9%**	**84.5%**	80.9%	80.0%

miniBUDE

- Proxy app created from University of Bristol BUDE protein simulator
  - calculates energy of each ligand-protein pair in different poses (position + rotation)
  - highly arithmetically intensive: FP arithmetic & trigonometric
- Kernel is triply-nested loop over proteins, ligands, poses
  - Chapel follows CUDA decomposition: 1D kernel assigning multiple poses to thread

```
foreach group in 0..<nposes/PPWI { ----------  fasten_main<PPWI><<<global, local, shared>>>( //
  for il in 0..<natlig {                                 p.natlig(), p.natpro(), protein, ligand, //
    for ip in 0..<natpro {                               transforms_0, transforms_1, transforms_2, //
      for param i in 0:int(32)..<PPWI {                  transforms_3, transforms_4, transforms_5, //
…                                                        results, forcefield, p.nposes());
```

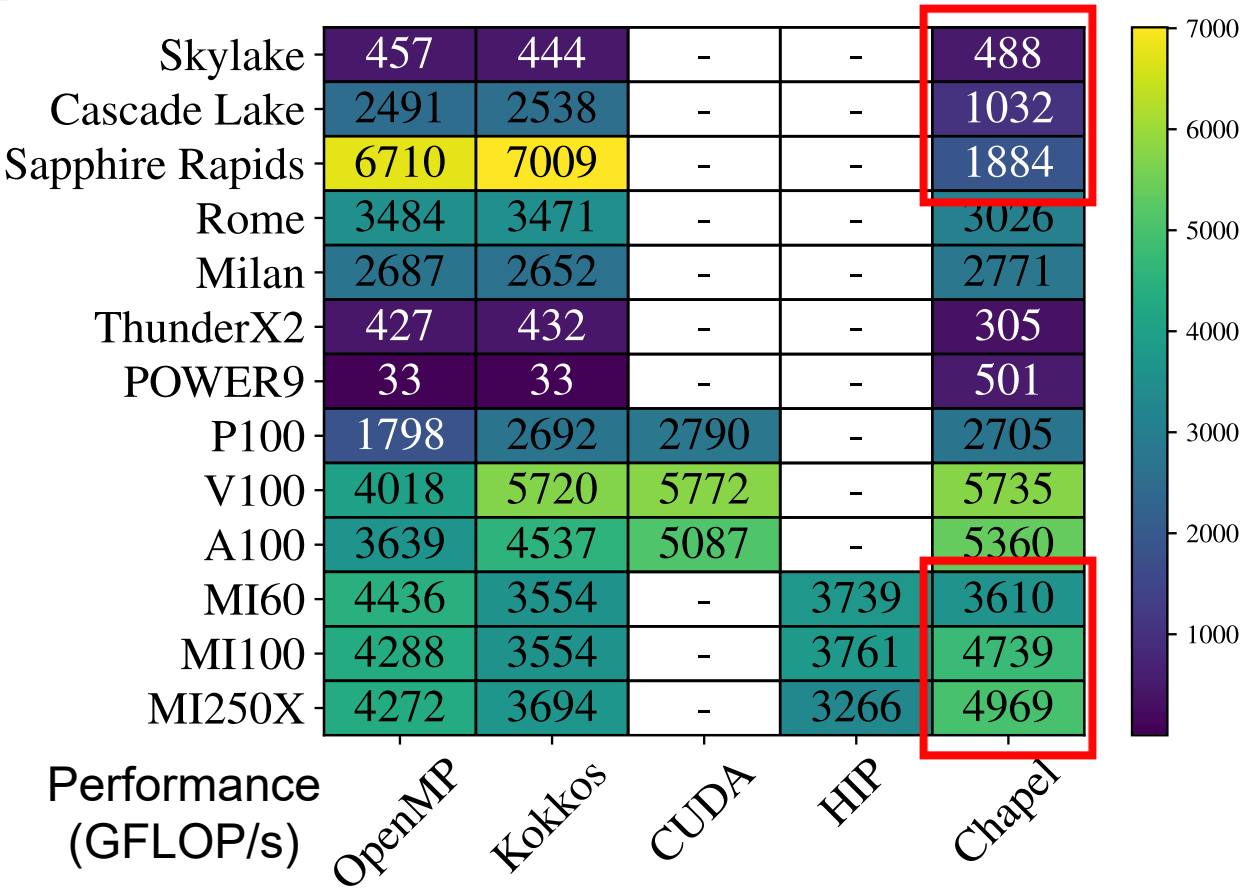
  - requires transfer of protein, ligand, and pose data to GPU; energies to host (in Chapel, these are simple array assignments)

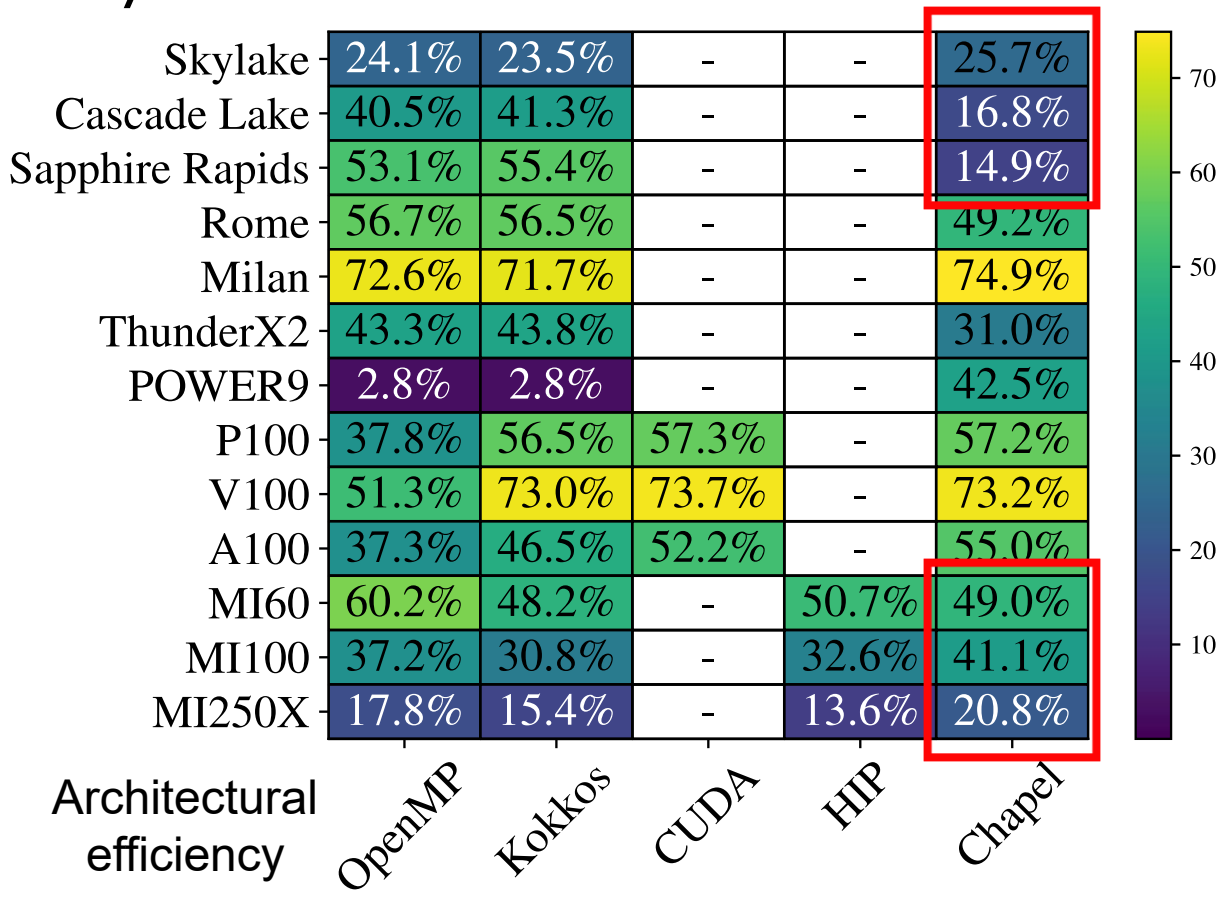
```
const protein = context.protein;
const ligand = context.ligand;
const forcefield = context.forcefield;
const poses: [0:int(32)..<6:int(32), 0..#nposes] real(32) = context.poses[{0..<6, gpuID*nposes..#nposes}];
```

https://github.com/milthorpe/miniBUDE/tree/v2

miniBUDE Performance Portability

miniBUDE v2 – small 'bm1' input





Platforms	OpenMP	Kokkos	CUDA	HIP	Chapel
All platforms*	43.0%	**44.8%**	0	0	33.8%
Supported CPUs*	43.0%	**43.1%**	0	0	25.9%
Supported GPUs	43.0%	47.1%	**60.2%**	39.7%	**53.1%**

* Except POWER9

TeaLeaf

- Collection of iterative sparse linear solvers, simulating heat conduction over time using five-point stencils over 2D grid
- Low arithmetic intensity = better suited to low STREAM balance
- 2D index domains: expose parallelism over both loops

```
#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];
  }
}
```

```
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);
      }
    });
```

```
[(i,j) in Domain.expand(-halo_depth)] p[i,j] = beta * p[i,j] + r[i,j];
```

https://github.com/milthorpe/TeaLeaf

S. McIntosh-Smith, et al., Tealeaf: A mini-application to enable design-space explorations for iterative sparse linear solvers. IEEE International Conference on Cluster Computing (CLUSTER), 2017.

TeaLeaf - Reductions

- Many sum reductions to compute global deltas or error metrics
  - In Chapel 2.0, these must be computed in global memory

```
Kokkos::parallel_reduce(
    x * y,
    KOKKOS_LAMBDA(const int &index, double &rrn_temp) {
      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) {
        u(index) += alpha * p(index);
        r(index) -= alpha * w(index);
        rrn_temp += r(index) * r(index);
      }
    },
    *rrn);
```

Chapel 2.0

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

Chapel 2.x

```
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;
}
```

TeaLeaf – Chapel Multi-Dimensional Indexing

- Using 2D indices improved readability of Chapel code and performed well on CPU platforms
- However, using 2D domains reduced GPU performance due to under-utilization of available GPU cores in Chapel 2.0
  - first dimension is assigned to GPU threads
  - remaining dimensions implemented as loops inside GPU kernel
- We replaced multi-dimensional loops with 1D loop over linearized space to allow full utilization of GPU cores

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

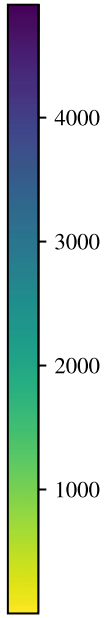
```
const Domain = {0..<y, 0..<x};
const OneD = {0..<y*x};
foreach oneDIdx in OneD {
  const ij = local_domain.orderToIndex(oneDIdx);
  u[ij] = energy[ij] * density[ij];
}
```

TeaLeaf Performance Portability

tea_bm_5.in – 4000×4000 CG solve, 10 iters

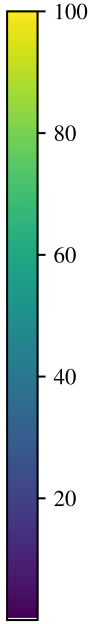
Runtime (s)

	OpenMP	Kokkos	CUDA	HIP	Chapel	Chapel 2D
Skylake	491	720	-	-	466	464
Cascade Lake	316	471	-	-	324	324
Sapphire Rapids	95	239	-	-	118	116
Rome	149	530	-	-	165	166
Milan	246	770	-	-	269	265
ThunderX2	413	864	-	-	400	403
POWER9	743	4781	-	-	361	360
P100	281	246	177	-	2574	4133
V100	149	129	96	-	544	3540
A100	131	69	50	-	314	3430
MI60	164	115	-	251	297	4911
MI100	148	111	-	233	231	2310
MI250X	143	79	-	73	169	2034



Application efficiency

	OpenMP	Kokkos	CUDA	HIP	Chapel	Chapel 2D
Skylake	94.4%	64.4%	-	-	99.5%	100%
Cascade Lake	100%	67.1%	-	-	97.7%	97.7%
Sapphire Rapids	100%	39.9%	-	-	80.5%	81.9%
Rome	100%	28.1%	-	-	90.4%	89.7%
Milan	100%	31.9%	-	-	91.4%	92.8%
ThunderX2	96.9%	46.3%	-	-	100%	99.2%
POWER9	48.4%	7.5%	-	-	99.8%	100%
P100	63.1%	72.0%	100%	-	6.9%	4.3%
V100	64.3%	74.3%	100%	-	17.6%	2.7%
A100	38.1%	72.5%	100%	-	15.9%	1.5%
MI60	70.3%	100%	-	45.9%	38.8%	2.3%
MI100	75.4%	100%	-	47.8%	48.1%	4.8%
MI250X	51.2%	93.1%	-	100%	43.4%	3.6%



Platforms	OpenMP	Kokkos	CUDA	HIP	Chapel	Chapel 2D
All platforms	**69.8%**	37.3%	0	0	31.5%	5.7%
Supported CPUs	85.8%	25.3%	0	0	93.7%	**94.0%**
Supported GPUs	57.3%	**83.5%**	**100.0%**	56.9%	17.8%	2.7%

Conclusions

- Pennycook, Sewall and Lee's metric PP remains a useful lens for evaluating portable programming models and identifying areas of strength and weakness
- Performance portability of OpenMP and Kokkos continues to improve
- Chapel is a new option for performance-portable parallel programming
  - concise code
  - (mostly) good performance across a wide range of platforms
  - easier path to multi-device, multi-node distribution
- Some issues remain with Chapel GPU code generation
  - fixing these will avoid performance pitfalls for users

Acknowledgments

- This research used resources of the Experimental Computing Laboratory (ExCL) and the Oak Ridge Leadership Computing Facility at 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.
- Thanks to 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.