Device Agnostic Programming with Chapel

NCI Workshop
3 July 2024
Josh Milthorpe

Includes content from
University of Bristol HPC Group OpenMP Tutorial
and
HPE Chapel Tutorials

Australian National University

Heterogeneity Everywhere

OLCF Frontier
AMD CPU+GPU

ALCF Aurora
Intel CPU+GPU

Bristol Isambard-AI
NVIDIA Grace-Hopper

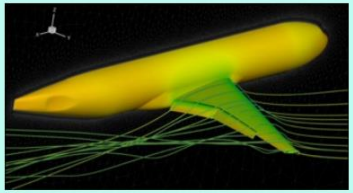
NCI Gadi
Intel CPU, NVIDIA GPU

Device-Agnostic Programming

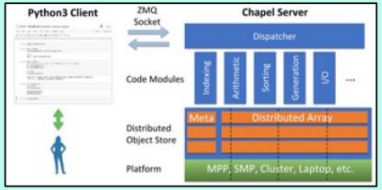
- Three main GPU vendors
  - NVIDIA
  - AMD
  - Intel
- Not to mention CPU
  - Intel / AMD
  - ARM
  - RISC-V
- We need a (performance-)portable way to program them all
  - OpenMP
  - OpenACC
  - SYCL
  - Kokkos
  - **Chapel**

Chapel Programming Language

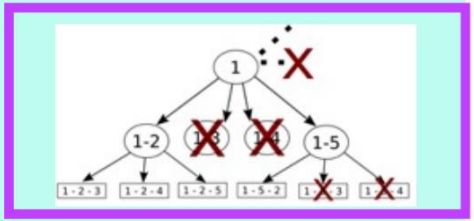
- General-purpose parallel programming language supporting productive app development, including:
  - data exploration
  - multi-physics CFD
  - computational astrophysics
- Ease of programming: first-class language features for task & data parallelism, synchronization, distributed memory
- Portability: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



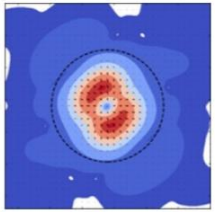
CHAMPS: 3D Unstructured CFD
Laurendeau, Bourgault-Côté, Parenteau, Plante, et al.
École Polytechnique Montréal



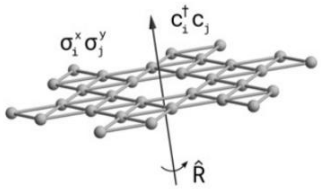
Arkouda: Interactive Data Science at Massive Scale
Mike Merrill, Bill Reus, et al.
U.S. DoD



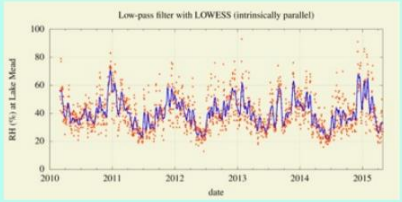
ChOp: Chapel-based Optimization
T. Carneiro, G. Helbecque, N. Melab, et al.
INRIA, IMEC, et al.



ChplUltra: Simulating Ultralight Dark Matter
Nikhil Padmanabhan, J. Luna Zagorac, et al.
Yale University et al.

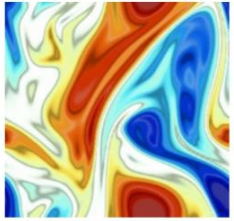


Lattice-Symmetries: a Quantum Many-Body Toolbox
Tom Westerhout
Radboud University

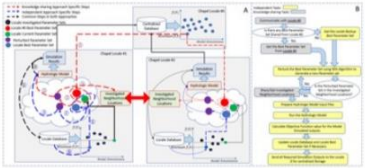


Desk dot chpl: Utilities for Environmental Eng.
Nelson Luis Dias
The Federal University of Paraná, Brazil

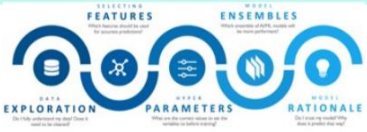
RapidQ: Mapping Coral Biodiversity
Rebecca Green, Helen Fox, Scott Bachman, et al.
The Coral Reef Alliance



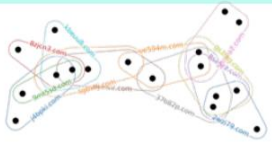
ChapQG: Layered Quasigeostrophic CFD
Ian Grooms and Scott Bachman
University of Colorado, Boulder et al.



Chapel-based Hydrological Model Calibration
Marjan Asgari et al.
University of Guelph



CrayAI HyperParameter Optimization (HPO)
Ben Albrecht et al.
Cray Inc. / HPE



CHGL: Chapel Hypergraph Library
Louis Jenkins, Cliff Joslyn, Jesun Firoz, et al.
PNNL

?

Your Application Here?

Active GPU efforts

Source: HPE (images provided by their respective teams and used with permission)

Chapel Resources

https://chapel-lang.org/

https://chapel-lang.org/docs/

Home

What is Chapel?
What's New?

Blog

Upcoming Events
Job Opportunities

How Can I Learn Chapel?

Contributing to Chapel
Community

Download / Install Chapel
Try Chapel Online

Documentation
Release Notes

Performance
Powered by Chapel

Presentations
Papers / Publications
Tutorials

ChapelCon
CHUG

Contributors / Credits

chapel+qs@discoursemail.com

The Chapel Parallel Programming Language

What is Chapel?

Chapel is a programming language designed for productive parallel computing at scale.

Why Chapel? Because it simplifies parallel programming through elegant support for:

- **data parallelism** to trivially use the cores of a laptop, cluster, or supercomputer
- **task parallelism** to create concurrency within a node or across the system
- **a global namespace** supporting direct access to local or remote variables
- **GPU programming** in a vendor-neutral manner using the same features as above
- **distributed arrays** that can leverage thousands of nodes' memories and cores

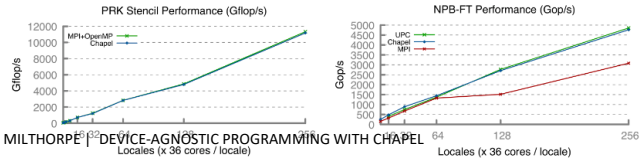
Chapel Characteristics

- **productive:** code tends to be similarly readable/writable as Python
- **scalable:** runs on laptops, clusters, the cloud, and HPC systems
- **fast:** performance competes with or beats conventional HPC programming models
- **portable:** compiles and runs in virtually any *nix environment
- **open-source:** hosted on GitHub, permissively licensed
- **production-ready:** used in real-world applications spanning diverse fields

New to Chapel?

As an introduction to Chapel, you may want to...

- watch an overview talk or browse its slides
- read a chapter-length introduction to Chapel
- learn about projects powered by Chapel
- check out performance highlights like these:



Chapel Documentation

version 2.1

Search docs

COMPILING AND RUNNING CHAPEL

Quickstart Instructions
Using Chapel
Platform-Specific Notes
Technical Notes
Tools
Docs for Contributors

WRITING CHAPEL PROGRAMS

Quick Reference
Hello World Variants
Primers
Language Specification
Standard Modules
Package Modules
Standard Layouts and Distributions
Mason Packages
Chapel Users Guide (WIP)

LANGUAGE HISTORY

Chapel Evolution

Chapel Documentation

Chapel Documentation

Compiling and Running Chapel

- Quickstart Instructions
- Using Chapel
- Platform-Specific Notes
- Technical Notes
- Tools
- Docs for Contributors

Writing Chapel Programs

- Quick Reference
- Hello World Variants
- Primers
- Language Specification
- Standard Modules
- Package Modules
- Standard Layouts and Distributions
- Mason Packages
- Chapel Users Guide (WIP)

Language History

- Chapel Evolution

Chapel Basics

Hello World in Chapel

- Fast prototyping

```
writeln("Hello, world!");
```

- “Production-grade”

```
module Hello {
    proc main() {
        writeln("Hello, world!");
    }
}
```

Hello World in Chapel: Configurable

- Fast prototyping (configurable):

```
config const audience = "world";
writeln("Hello, ", audience, "!");
```

- “Production-grade”

```
module Hello {
    config const audience = "world";
    proc main() {
        writeln("Hello, ", audience, "!");
    }
}
```

- To change “audience” for a given run:

```
> ./hello -saudience="mate“
Hello, mate!
```

Variables, Constants, and Parameters

- Basic syntax

```
declaration:
  var identifier [: type] [= init-expr];
  const identifier [: type] [= init-expr];
  param identifier [: type] [= init-expr];
```

- Examples

```
const pi: real = 3.14159;
var count: int; // initialized to 0
param debug = true; // inferred to be bool
```

- Meaning
  - `var`/`const`: execution-time variable/constant
  - `param`: compile-time constant
  - No `init-expr` : initial value is the type's default
  - No type : type is taken from `init-expr`
- Configuration variables
  - Value can be overridden on command-line (`param` = compile; or `const`/`var` = execution)

Hello World in Chapel: Task-Parallel

```
const numTasks = here.maxTaskPar;
  coforall tid in 1..#numTasks do
    writef("Hello from task %n of %n on %s\n",
      tid, numTasks, here.name);
```

'here' refers to the locale on which we're currently running

how many concurrent tasks does this node support (typically the number of processor cores)?

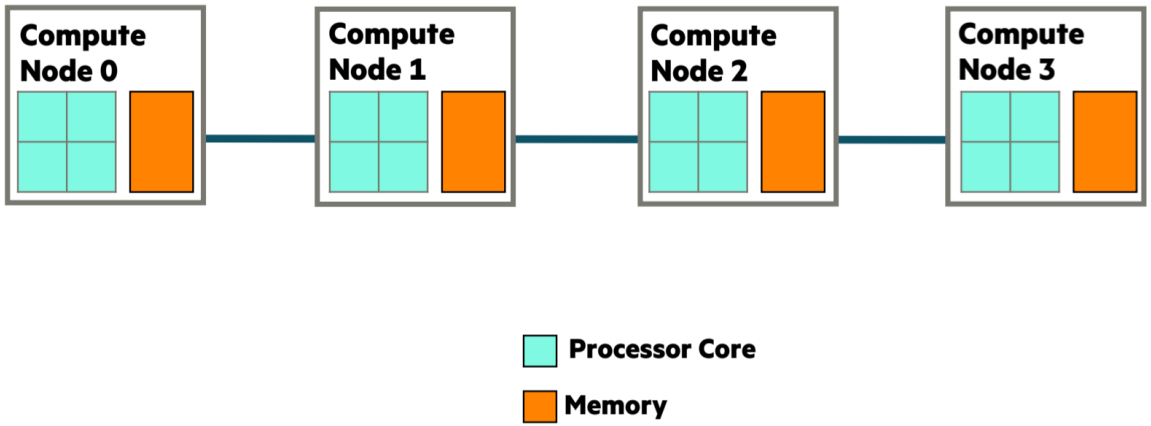
what's my locale's name?

a 'coforall' loop executes each iteration as an independent task

```
> chpl hello.chpl
> ./hello
Hello from task 2 of 48 on gadi-cpu-clx-3019.gadi.nci.org.au
Hello from task 13 of 48 on gadi-cpu-clx-3019.gadi.nci.org.au
Hello from task 39 of 48 on gadi-cpu-clx-3019.gadi.nci.org.au
Hello from task 15 of 48 on gadi-cpu-clx-3019.gadi.nci.org.au
…
```

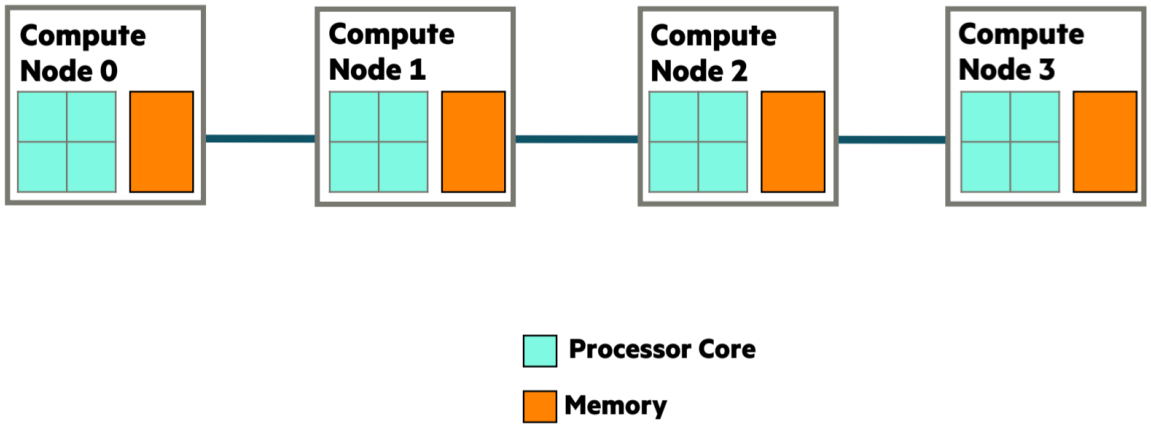
Locales

- In Chapel, a locale refers to a compute resource with…
  - processors, so it can run tasks
  - memory, so it can store variables
- For now, think of each compute node as having one locale run on it



Locales

- Two key built-in variables for referring to locales in Chapel programs:
  - `Locales`: An array of locale values representing the system resources on which the program is running
  - `here`: The locale on which the current task is executing



Locale Operations

- Locale methods support queries about the target system:

```
proc locale.physicalMemory(…) { … }
proc locale.maxTaskPar { … }
proc locale.id { … }
proc locale.name { … }
```

- *On-clauses* support placement of computations:

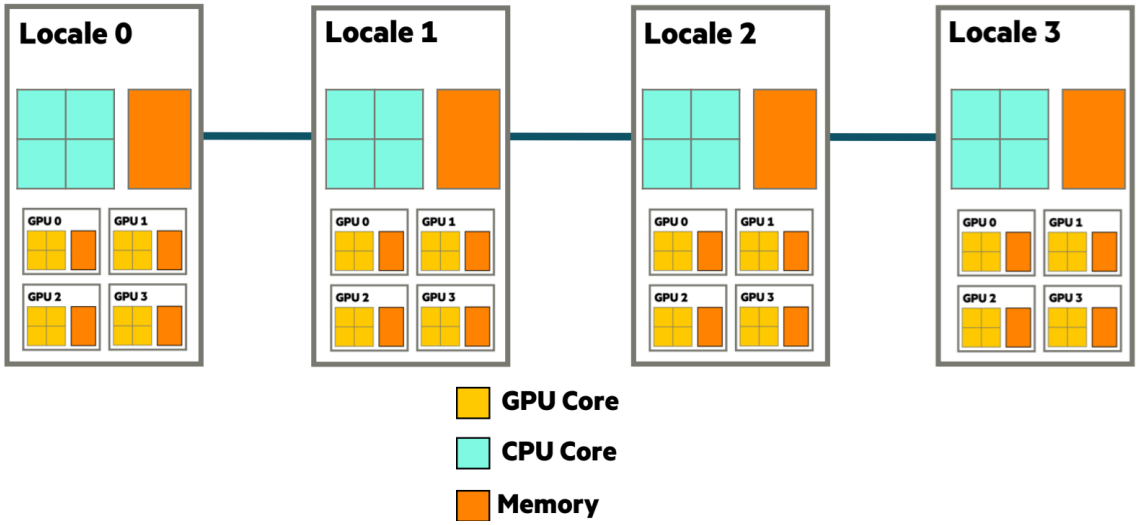
```
writeln("on locale 0");
on Locales[1] do
  writeln("now on locale 1");
writeln("on locale 0 again");
```

```
on A[i,j] do
  bigComputation(A);

on node.left do
  search(node.left);
```

Locales - GPUs

- Complicating matters, compute nodes may have GPUs with their own processors and memory
  - We represent these as *sub-locales* in Chapel



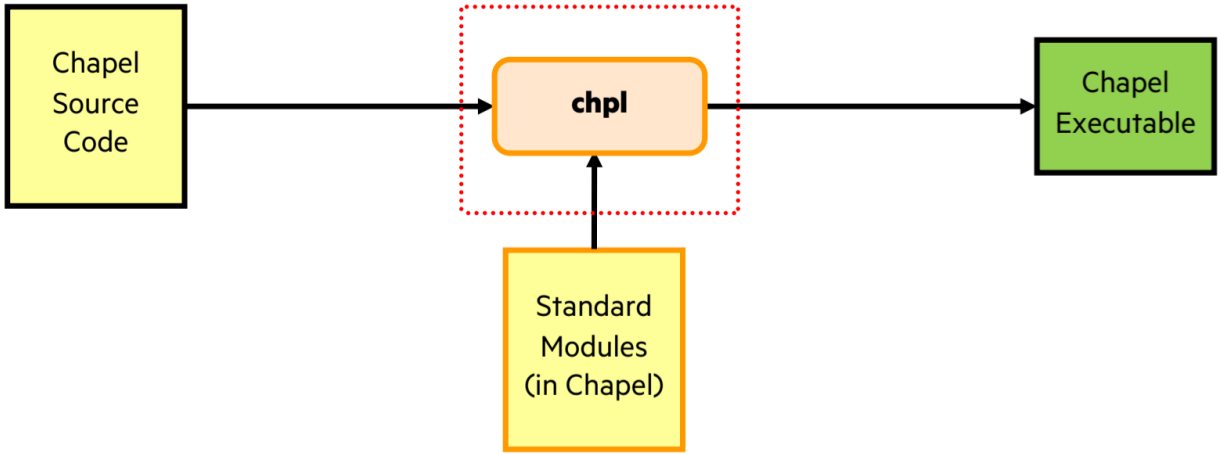
Hello World in Chapel: Task-Parallel + GPUs

```
const numTasks = here.maxTaskPar;
coforall tid in 1..#numTasks do
  writef("Hello from task %n of %n on %s\n",
    tid, numTasks, here.name);
  coforall gpu in here.gpus do on gpu {
    writef("Hello from GPU %s\n", here.name);
  }
```

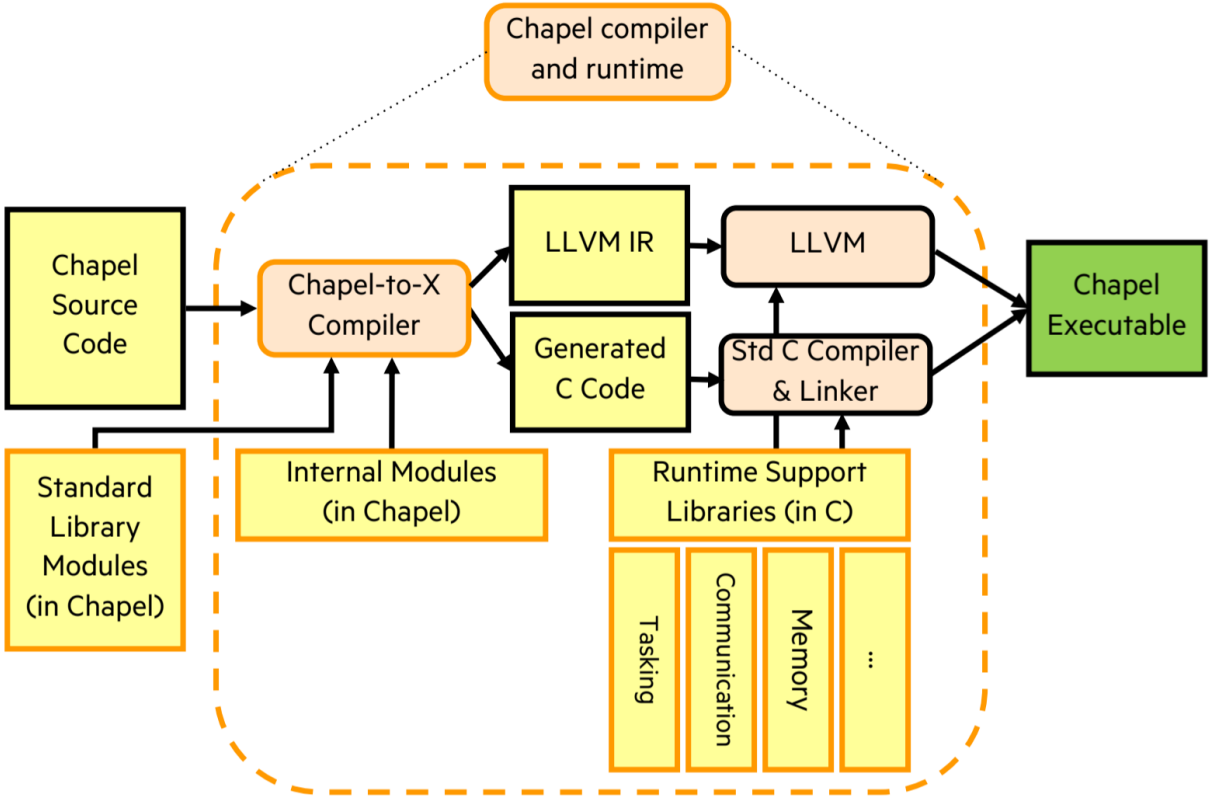
on clause moves task to GPU locale

Array of locales representing all the GPUs on this node

Compiling Chapel



Chapel Compiler Architecture



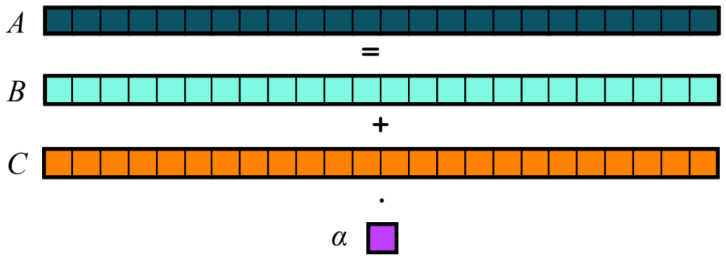
Data Parallelism

Stream Triad: Parallel Computation

Given: m-element vectors A, B, C

Compute: $\forall i \in 1..m,\ A_i = B_i + \alpha \cdot C_i$

In pictures:

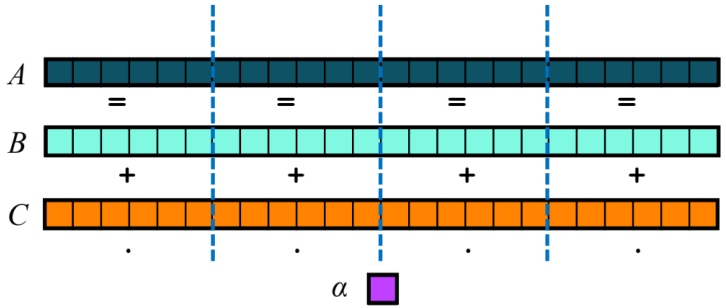


Stream Triad: Parallel Computation

Given: m-element vectors A, B, C

Compute: $\forall i \in 1..m,\ A_i = B_i + \alpha \cdot C_i$

In pictures (shared memory / multicore):

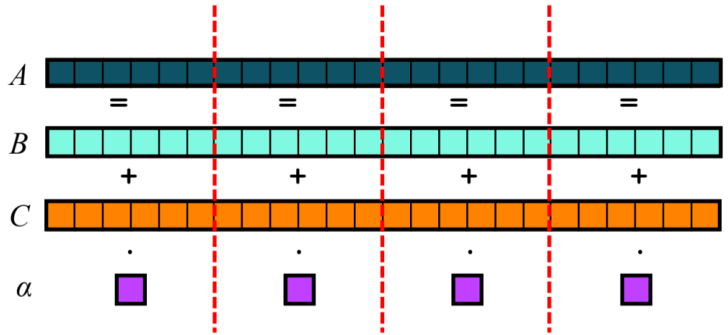


Stream Triad: Parallel Computation

Given: m-element vectors A, B, C

Compute: $\forall i \in 1..m,\ A_i = B_i + \alpha \cdot C_i$

In pictures (distributed memory):

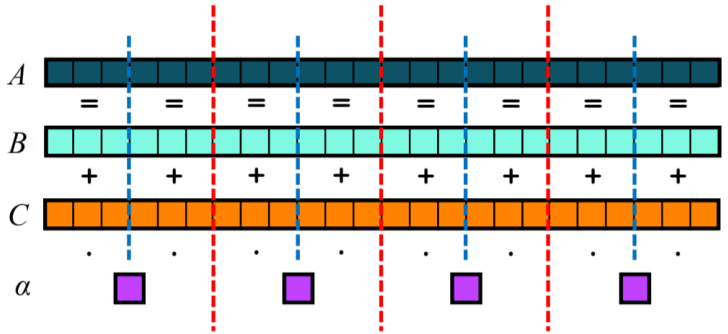


Stream Triad: Parallel Computation

Given: m-element vectors A, B, C

Compute: $\forall i \in 1..m,\ A_i = B_i + \alpha \cdot C_i$

In pictures (distributed memory multicore):



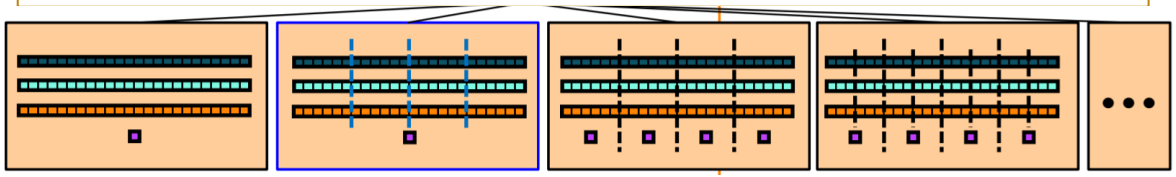
Stream Triad: Chapel

```
use BlockDist;
config const m = 1000,
             alpha = 3.0;

const ProblemSpace = blockDist.createDomain({1..m});
var A, B, C: [ProblemSpace] real;

B = 2.0;
C = 1.0;
A = B + alpha * C;
```

The special sauce:
How should this index set — and any arrays and computations over it— be mapped to the system?



Philosophy: Good, top-down language design can tease system-specific implementation details away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.

Tuples

- support lightweight grouping of values
  - e.g., passing/returning multiple procedure arguments at once
  - short vectors
  - multidimensional array indices
- support heterogeneous data types

```
var coord: (int, int, int) = (1, 2, 3);
var coordCopy: 3*int = coord;
var (i1, i2, i3) = coord;
var triple: (int, string, real) = (7, "eight", 9.0);
```

Domains

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D with (ref A) do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

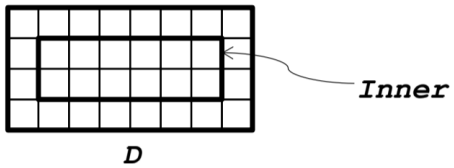
Domain

```
> chpl dataParallel.chpl
> ./dataParallel -nl 1 --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Domain:

- A first-class index set
- The fundamental Chapel concept for data parallelism

```
config const m = 4, n = 8;
const D = {1..m, 1..n};
const Inner = {2..m-1, 2..n-1};
```



Arrays

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D with (ref A) do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

Array

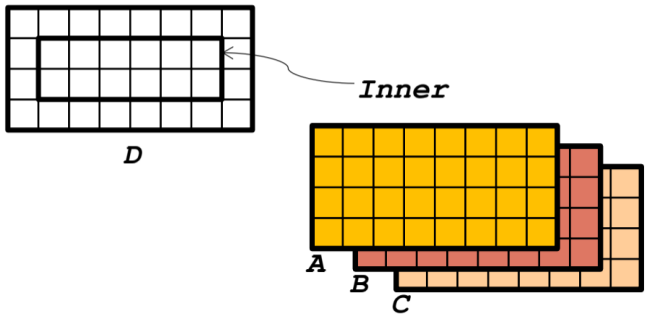
Array:

- Collection of data items of the same type, indexed by domain

```
> chpl dataParallel.chpl
> ./dataParallel -nl 1 --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Arrays

```
config const m = 4, n = 8;
const D = {1..m, 1..n};
const Inner = {2..m-1, 2..n-1};
var A, B, C: [D] real;
```



Data-Parallel Forall Loops

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D with (ref A) do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

Data-parallel loop

```
> chpl dataParallel.chpl
> ./dataParallel -nl 1 --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Forall loops: Central concept for data parallel computation

- Like for-loops, but parallel
- Implementation details determined by iterand (e.g., D below)
  - specifies number of tasks, mapping of iterations to tasks, …
  - in practice, typically uses a number of tasks appropriate for target HW

```
forall (i,j) in D with (ref A) do
  A[i,j] = i + j/10.0;
```

1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8
2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8
3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8
4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8

Forall loops assert…

…**parallel safety**: OK to execute iterations simultaneously

…**order independence**: iterations could occur in any order

…**serializability**: all iterations could be executed by one task

– i.e. can't have synchronization dependencies between iterations

Comparison of Loop Structures

- **For loops**: executed using one task
  - use when a loop must be executed serially
  - or when one task is sufficient for performance

- **Forall loops**: typically executed using 1 < #tasks << #iters
  - use when a loop *should* be executed in parallel…
  - …but can legally be executed serially
  - use when desired # tasks << # of iterations

- **Coforall loops**: executed using a task per iteration
  - use when the loop iterations *must be* executed in parallel
  - use when you want # tasks == # of iterations
  - use when each iteration has substantial work

Implicit Loops: Promotion of Scalar Subroutines and Array Operations

- Any function or operator that takes scalar arguments can be called with array expressions instead

```
proc foo(x: real, y: real, z: real) {
  return x**y + 10*z;
}
```

- Interpretation

```
C = foo(A, 2, B);
```

is equivalent to:

```
forall (c, a, b) in zip(C, A, B) do
  c = foo(a, 2, b);
```

as is:

```
C = A**2 + 10*B;
```

Using Task Intents in Loops

- Procedure argument intents
  - Tell how to pass a symbol actual argument into a formal parameter
  - Default intent is `const`, which means formal can't be modified in procedure body
  - `ref` means formal can be changed AND that change will be visible elsewhere, e.g., at the call site
  - Others: `in`, `out`, and `inout` refer to copying the actual argument in, the formal out, or both
- Task intents in loops
  - Similar to argument intents in syntax and philosophy
  - Also have a `reduce` intent similar to OpenMP
  - `reduce` intent means each task has its own copy and specified operation like '+' will combine at end of loop
- Design principles
  - Avoid common race conditions
  - Avoid copies of (potentially) large data structures

Reduce Intents

- A variable used in a parallel construct marked with `reduce` intent ensures that each task accumulates to its own private copy
  - All task copies are reduced together on exit of the parallel construct
- Standard reductions supported by default:

  `+, *, min, max, &, |, &&, ||, minloc, maxloc, …`
- Reductions can target arbitrary iterable expressions:

```
const total = + reduce Arr,
      factN = * reduce 1..n,
      biggest = max reduce (forall i in myIter() do
foo(i));
```

- Similar story for `scan`

Task Intents in Forall Loops: Scalars

```
var sum: real;
forall i in 1..n do
    sum += computeMyResult(i);
```

Default intent of scalars is `const in` so this is illegal (and avoids a race)

```
var sum: real;
forall i in 1..n with (ref sum) do
    sum += computeMyResult(i);
```

With `ref` intent, we are requesting a race

```
var sum: real;
forall i in 1..n with (+ reduce sum) do
    sum += computeMyResult(i);
```

Override default intent so that each task accumulates its own copy. On loop exit, all tasks combine their results into original ‘sum’

Task Intents in Forall Loops: Arrays

```
var bucketCount: [0..<m] real;
forall i in 1..n with (ref bucketCount) do
  bucketCount[i % m] += 1;
```

`ref` intent avoids array copies, but can result in data races

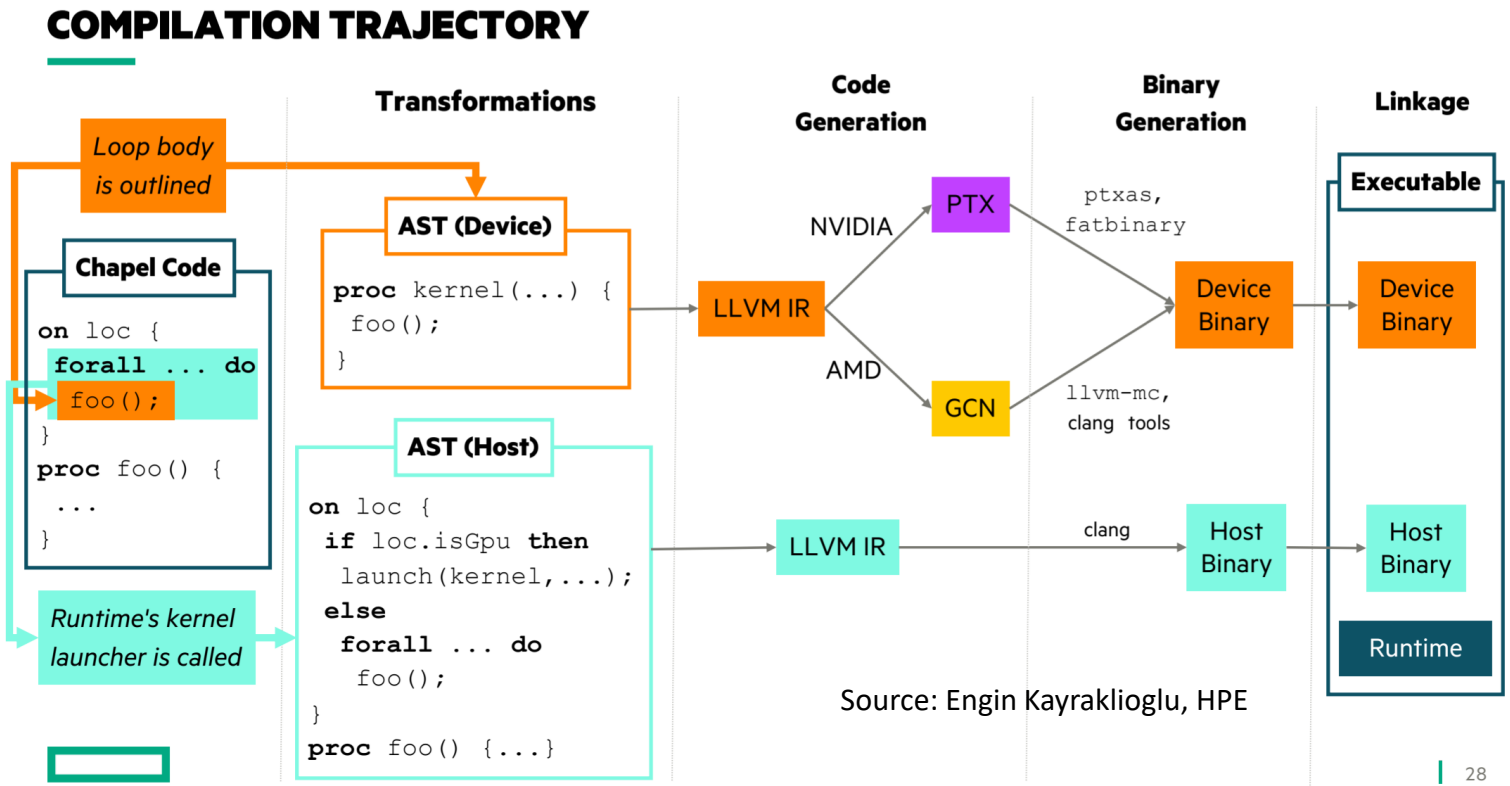
```
var bucketCount: [0..<m] real;
forall i in 1..n with (in bucketCount) do
    bucketCount[i % m] += 1;
```

`in` intent results in each task having its own copy

```
var bucketCount : [0..<m] real;
forall i in 1..n with (+ reduce bucketCount) do
  bucketCount[i % m] += 1;
```

`reduce` intent results in each task having own copy, but then on loop exit tasks combine their results into the original 'bucketCount' variable

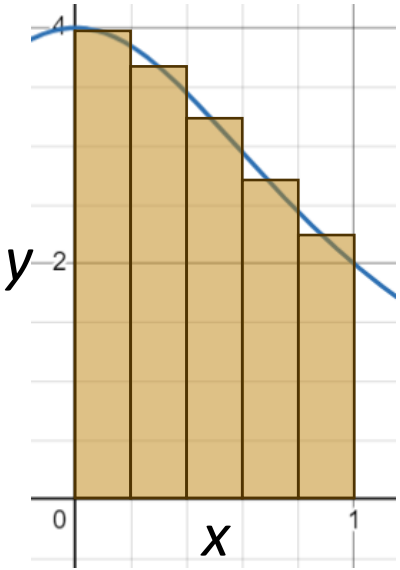
Chapel Compiler Architecture - GPU



Source: Engin Kayraklioglu, HPE

Exercise: Pi

Numerical Integration: The Pi Program



Mathematically, we know that:

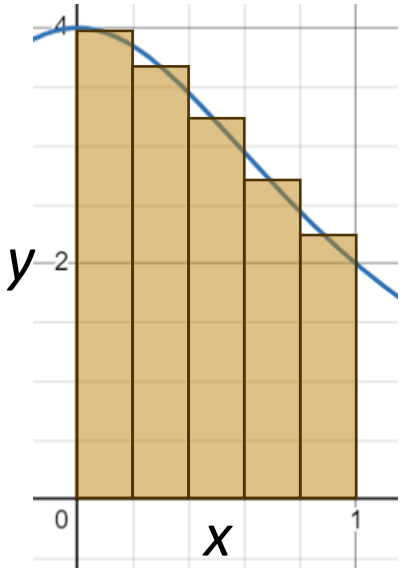
$$\int_0^1 \frac{4}{1-x^2} = \pi$$

We can approximate the integral using a Riemann sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the midpoint of interval i.

Serial Pi Program




```
// pi.chpl
const step: real = 1.0 / num_steps;
var sum: real = 0.0;
for i in 1..#num_steps {
    const x = (i - 0.5) * step;
    sum = sum + 4.0 / (1.0 + x * x);
}
const pi = step * sum;
```

Exercise: Heat

Five-Point Stencil: The Heat Program

- The heat equation models changes in temperature over time:

$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0$$

- We'll solve this numerically using a **finite difference** discretization.
- $u = u(t, x, y)$ is a function of space and time.
- Partial differentials are approximated using diamond difference formulae:

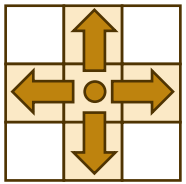
$$\frac{\partial u}{\partial t} \approx \frac{u(t+1, x, y) - u(t, x, y)}{dt}$$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(t, x+1, y) - 2u(t, x, y) + u(t, x-1, y)}{dx^2}$$

  - Forward difference in time, central finite difference in space.

Five-Point Stencil: The Heat Program

- Given an initial value of u , and any boundary conditions, we can calculate the value of u at time t+1 given the value at time t.
- Each update requires only the central value and the north, south, east and west neighbours:

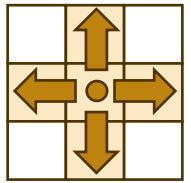


- Computation is essentially a weighted average of each cell and its neighbouring cells.
- If on a boundary, look up a boundary condition instead.

Five-Point Stencil: Solve Kernel

```
proc solve(…) {
  // Finite difference constant multiplier
  const r = alpha * dt / (dx*dx);
  const r2 = 1.0 - 4.0*r;

  // Loop over the nxn grid
  forall (i,j) in outerDom {
    // Update the 5-point stencil, using boundary conditions on the edges of the domain.
    // Boundaries are zero because the MMS solution is zero there.
    u_tmp[i,j] =  r2 * u[i,j] +
    r * (if i < n-1 then u[i+1,j] else 0.0) +
    r * (if i > 0   then u[i-1,j] else 0.0) +
    r * (if j < n-1 then u[i,j+1] else 0.0) +
    r * (if j > 0   then u[i,j-1] else 0.0);
  }
}
```



Performance Portability

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?

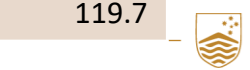
S. J. Pennycook, J. D. Sewall, and V. W. Lee, Implications of a metric for performance portability, Future Generation Computer Systems, vol. 92, pp. 947–958, 2019.

Mini-apps

- We created new Chapel implementations of three mini-apps developed by the University of Bristol's High Performance Computing group
- These mini-apps 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

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



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

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);
```

```
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 2.0

```
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;
}
```

Chapel 2.1

TeaLeaf – Chapel Multi-Dimensional Indexing

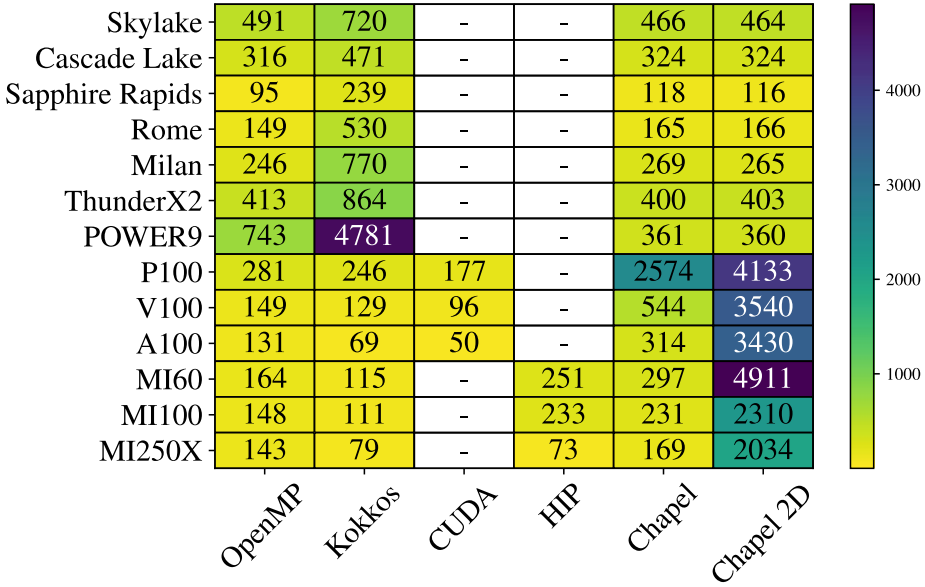
- Using 2D indices improved readability of Chapel code and performed well on CPU platforms
- However, using 2D domains reduced GPU performance due to under-utilization of available GPU cores in Chapel 2.0
  - first dimension is assigned to GPU threads
  - remaining dimensions implemented as loops inside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

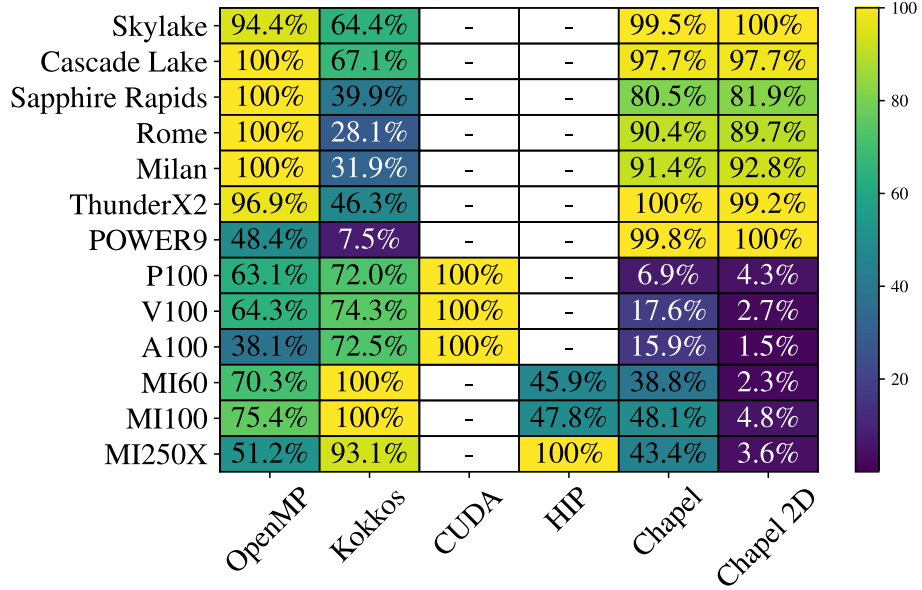
```
const Domain = {0..<y, 0..<x};
forall ij in Domain {
  u[ij] = energy[ij] * density[ij];
}
```

```
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





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%