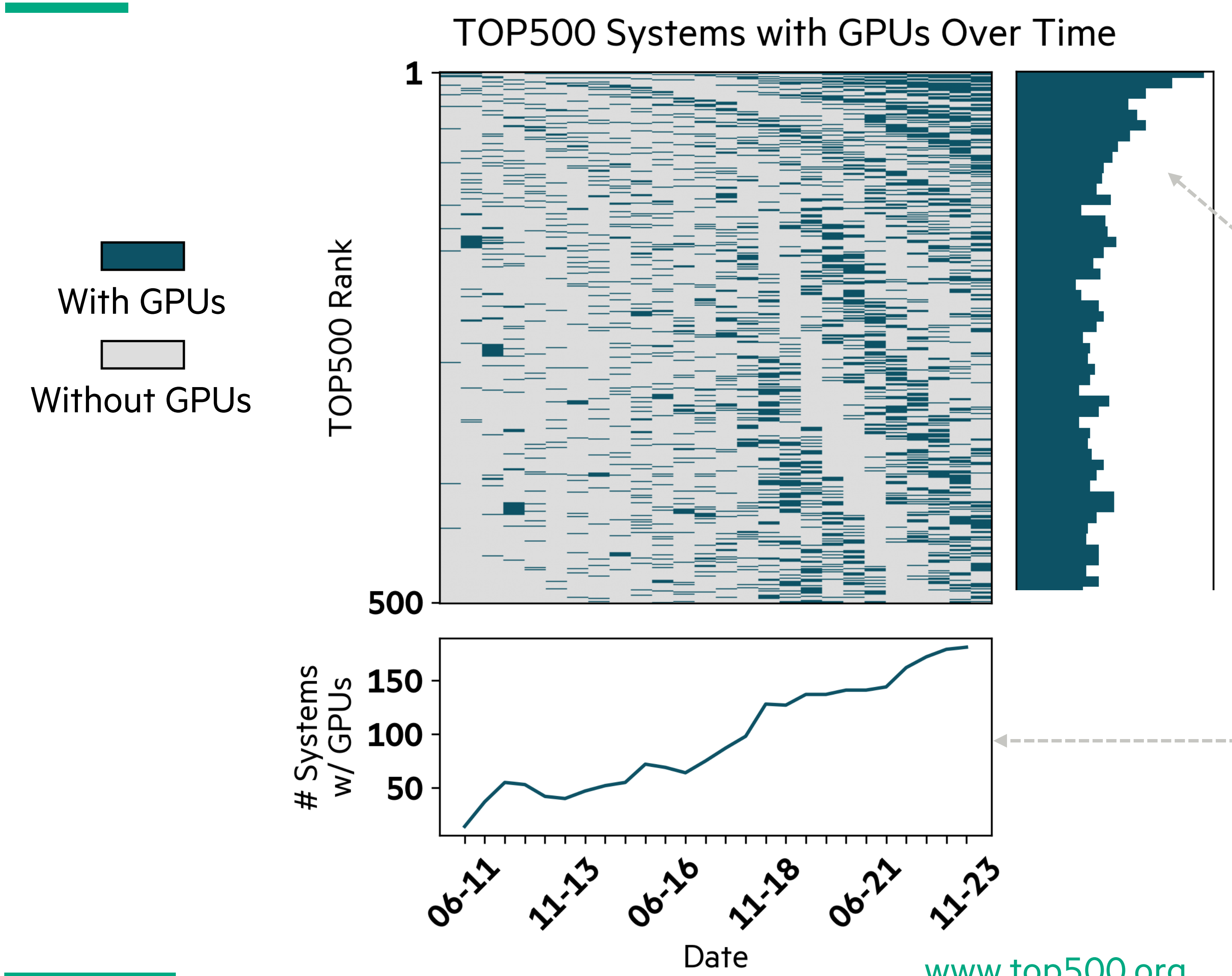
Hewlett Packard Enterprise

MAKING PARALLEL PROGRAMMING AND GPUS MORE ACCESSIBLE WITH CHAPEL

Engin Kayrakliogu
May 31st, 2024

engin@hpe.com
linkedin.com/in/engink

IT IS HARD TO AVOID GPUS IN HPC



www.top500.org

41% of **top 10** have GPUs

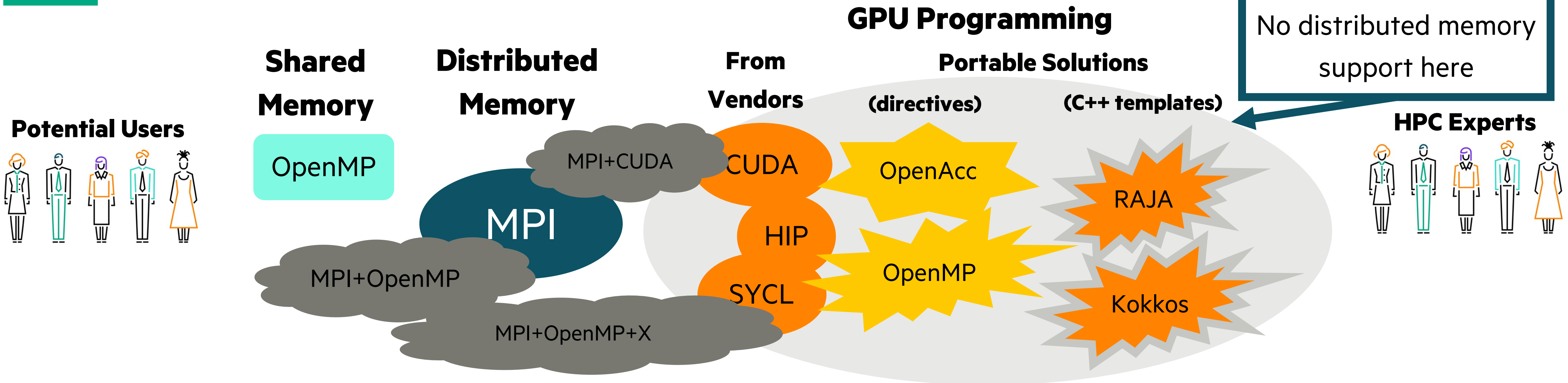
72% in the last **3 years**

27% of **top 100** have GPUs

52% in the last **3 years**

13x more systems w/GPUs

GPUS ARE EASY TO FIND… BUT DIFFICULT TO PROGRAM



All are effective, powerful, essential and tested technologies!

- … but programming for multiple nodes with GPUs appears to require at least 2 programming models
  - all of the models rely on C/C++/Fortran, which are different than the languages being taught these days
  - as a result, *using GPUs in HPC has a high barrier of entry*

Chapel is an alternative for productive distributed/shared memory GPU programming in a vendor-neutral way.

WHAT IS CHAPEL?

Chapel: A modern parallel programming language

- portable & scalable
- open-source & collaborative

Goals:

- Support general parallel programming
- Make parallel programming at scale far more productive

chapel-lang.org

WHAT IS CHAPEL?

Chapel works everywhere

- you can develop on your laptop and have the code scale on a supercomputer
- GPUs can be targeted in a vendor-neutral way
- runs on Linux laptops/clusters, Cray systems, MacOS, WSL, AWS, Raspberry Pi
- shown to scale on Cray networks (Slingshot, Aries), InfiniBand, RDMA-Ethernet

Chapel makes distributed/shared memory parallel programming easy

- data-parallel, locality-aware loops,
- ability to move execution and allocation to remote nodes,
- distributed arrays and bulk array operations
- different types of parallelism can be expressed with the same language features

WHAT IS CHAPEL?

Chapel works everywhere

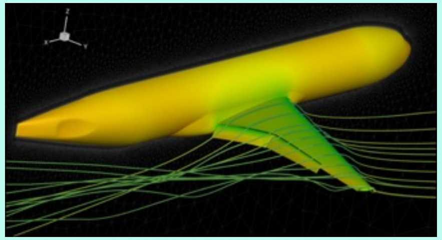
- you can develop on your laptop and have the code scale on a supercomputer
- GPUs can be targeted in a vendor-neutral way
- runs on Linux laptops/clusters, Cray systems, MacOS, WSL, AWS, Raspberry Pi
- shown to scale on Cray networks (Slingshot, Aries), InfiniBand, RDMA-Ethernet

Chapel makes distributed/shared memory parallel programming easy

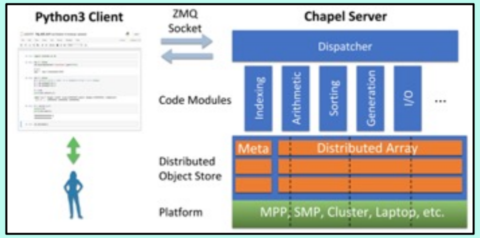
- data-parallel, locality-aware loops,
- ability to move execution and allocation to remote nodes,
- distributed arrays and bulk array operations
- different types of parallelism can be expressed with the same language features

APPLICATIONS OF CHAPEL

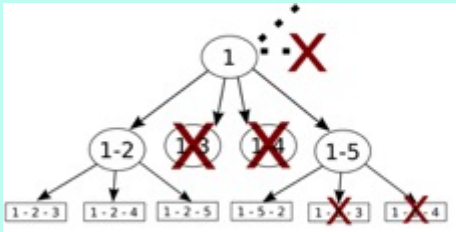
Active GPU efforts



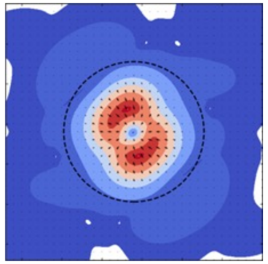
CHAMPS: 3D Unstructured CFD
Laurendeau, Bourgault-Côté, Parenteau, Plante, et al.
École Polytechnique Montréal



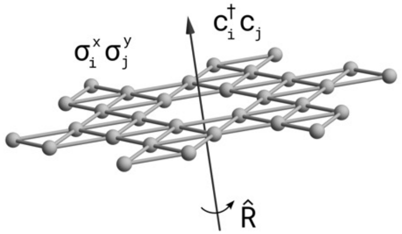
Arkouda: Interactive Data Science at Massive Scale
Mike Merrill, Bill Reus, et al.
U.S. DoD



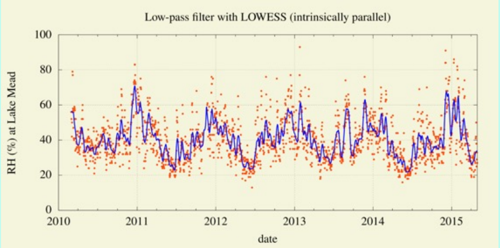
ChOp: Chapel-based Optimization
T. Carneiro, G. Helbecque, N. Melab, et al.
INRIA, IMEC, et al.



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

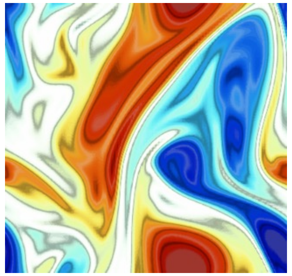


Lattice-Symmetries: a Quantum Many-Body Toolbox
Tom Westerhout
Radboud University

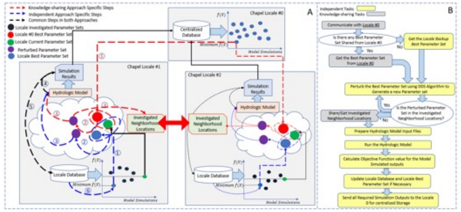


Desk dot chpl: Utilities for Environmental Eng.
Nelson Luis Dias
The Federal University of Paraná, Brazil

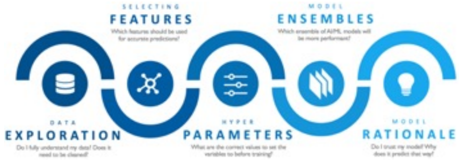
RapidQ: Mapping Coral Biodiversity
Rebecca Green, Helen Fox, Scott Bachman, et al.
The Coral Reef Alliance



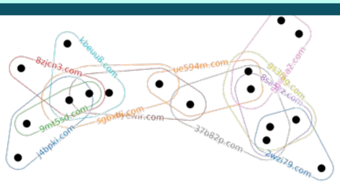
ChapQG: Layered Quasigeostrophic CFD
Ian Grooms and Scott Bachman
University of Colorado, Boulder et al.



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?

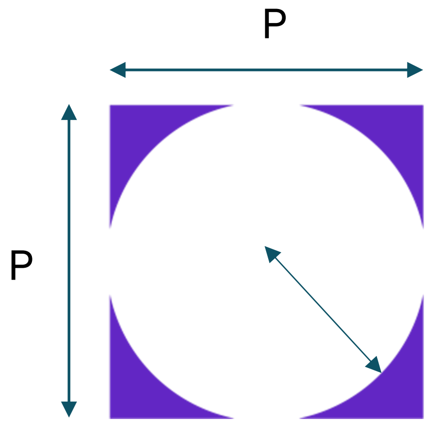
(images provided by their respective teams and used with permission)

CORAL REEF SPECTRAL BIODIVERSITY

1. Read in a (M x N) raster image of habitat data

2. Create a (P x P) mask to find all points within a given radius.

3. Convolve this mask over the entire domain and perform a weighted reduce at each location.

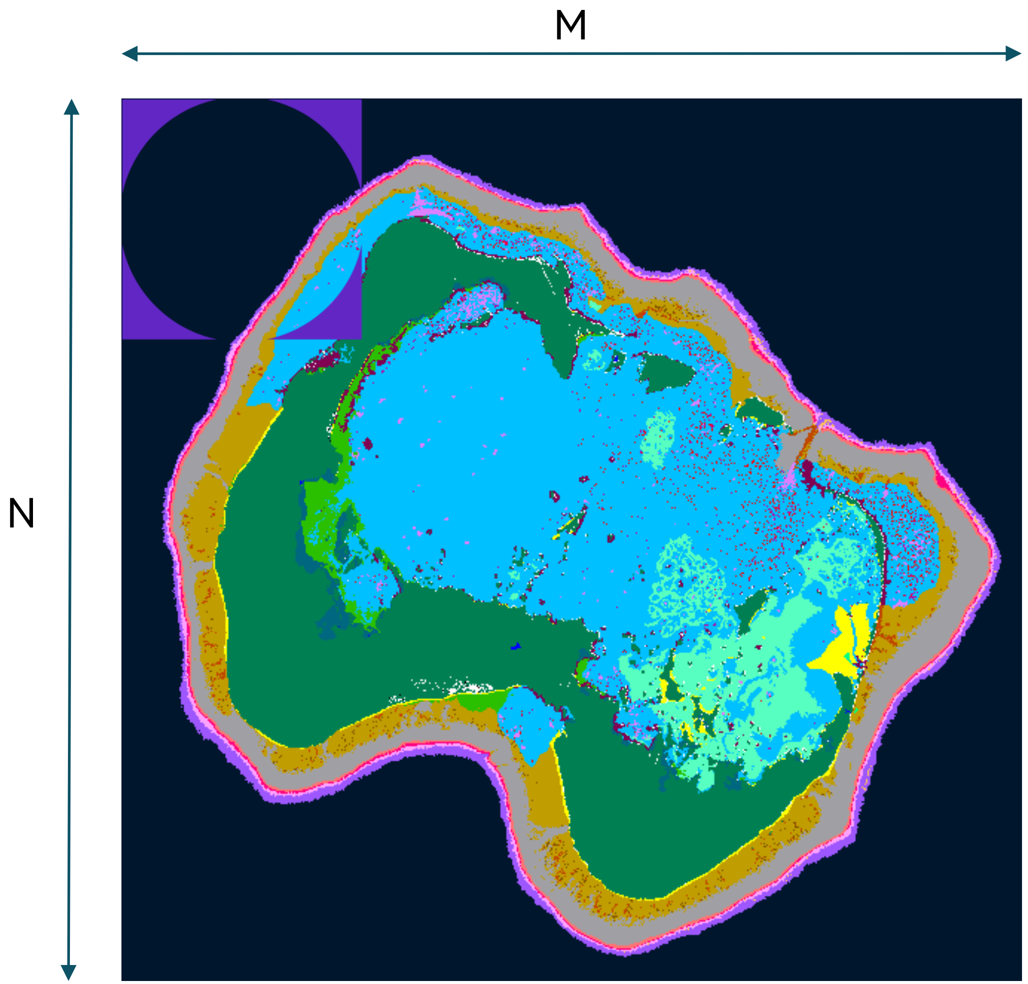


Algorithmic complexity: $O(MNP^3)$

Typically:

- M, N > 10,000

- P ~ 400



CORAL REEF SPECTRAL BIODIVERSITY

```
proc convolve(InputArr, OutputArr) {    // 3D Input, 2D Output
 for ... {
  tonOfMath();
 }
}
proc main() {
 var InputArr: ...;
 var OutputArr: ...;

 convolve(InputArr, OutputArr);
}
```

CORAL REEF SPECTRAL BIODIVERSITY

```
proc convolve(InputArr, OutputArr) {    // 3D Input, 2D Output
 foreach ... {
  tonOfMath();
 }
}
proc main() {
 var InputArr: ...;
 var OutputArr: ...;

 coforall loc in Locales do on loc {          // use all nodes in parallel...
  coforall gpu in here.gpus do on gpu {       // using GPUs on this node in parallel...
   coforall task in 0..#numWorkers {          // using numWorkers on this GPU in parallel.
    var MyInputArr = InputArr[...];
    var MyOutputArr: ...;
    convolve(MyInputArr, MyOutputArr);
    OutputArr[...] = MyOutputArr;
}}}}
```

Using a different loop flavor to enable GPU execution.

Multi-node, multi-GPU, multi-thread parallelism are expressed using the same language constructs.

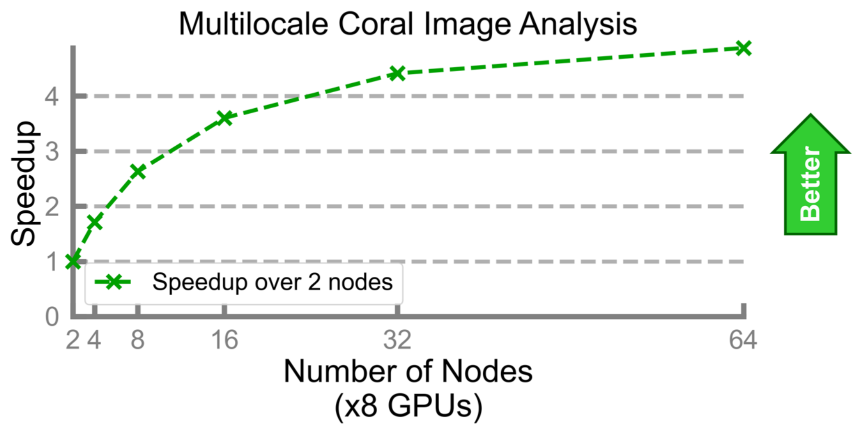
High-level, intuitive array operations work across nodes and/or devices

CORAL REEF SPECTRAL BIODIVERSITY

```
proc convolve(InputArr, OutputArr) {   // 3D Inp
  foreach ... {
   tonOfMath();
  }
}
proc main() {
  var InputArr: ...;
  var OutputArr: ...;

  coforall loc in Locales do on loc {        // u
   coforall gpu in here.gpus do on gpu {   // u
    coforall task in 0..#numWorkers { // using pa
     var MyInputArr = InputArr[...];
     var MyOutputArr: ...;
     convolve(MyInputArr, OutputArr);
     OutputArr[...] = MyOutputArr;
}}}}
```

Runs on Frontier!

- 5x improvement going from 2 to 64 nodes
  - (from 16 to 512 GPUs)
- Straightforward code changes:
  - from sequential Chapel code
  - to GPU-enabled one
  - to multi-node, multi-GPU, multi-thread



- Scalability improvements coming soon!

WHAT WE WILL DISCUSS TODAY

- Native GPU programming in Chapel using simple snippets
- Two stories from the community analyzing performance

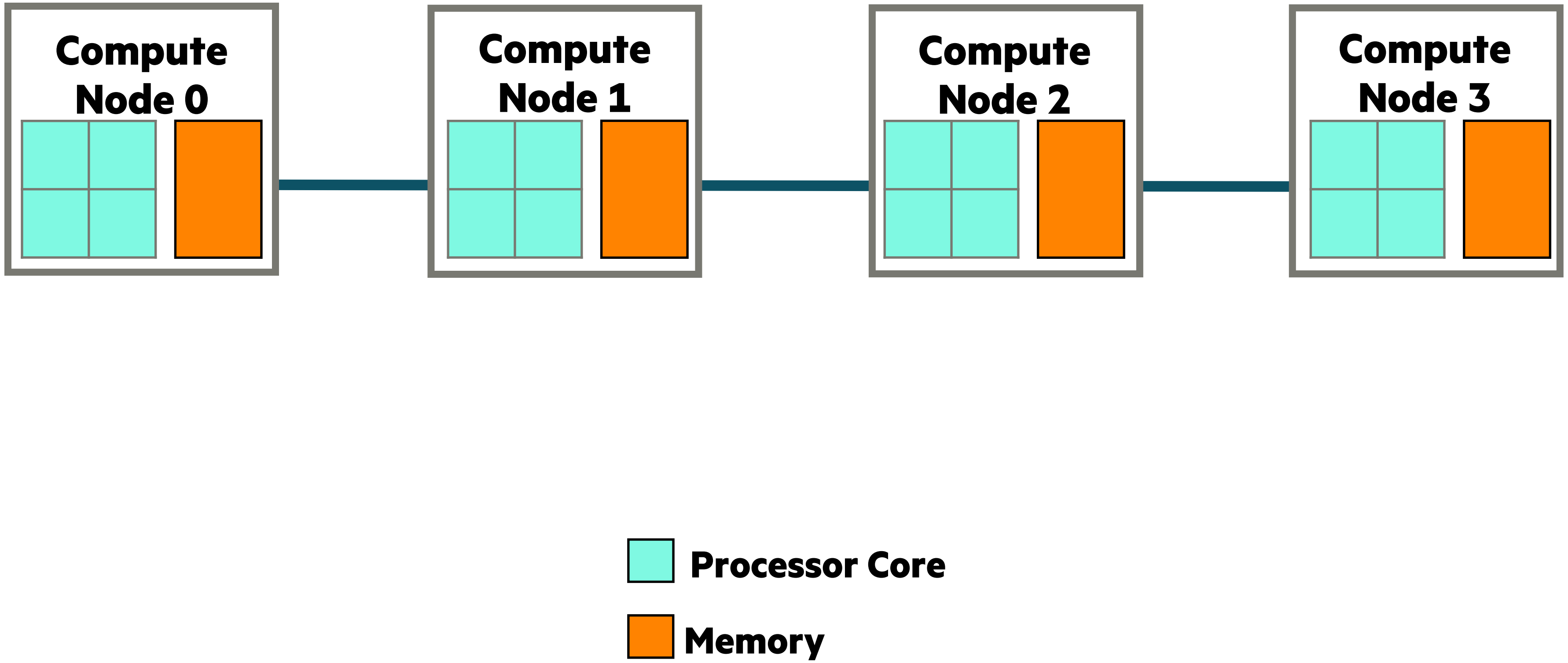
What we will not discuss today:

- Comprehensive list of Chapel features
  - (important ones will be covered)
- How GPU support is implemented
  - (happy to go over some backup slides, if there's interest)
- Everything you can do with GPUs using Chapel
  - (there's only so much time ☺)

GPU PROGRAMMING IN CHAPEL

LOCALES IN CHAPEL

- 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 being a locale

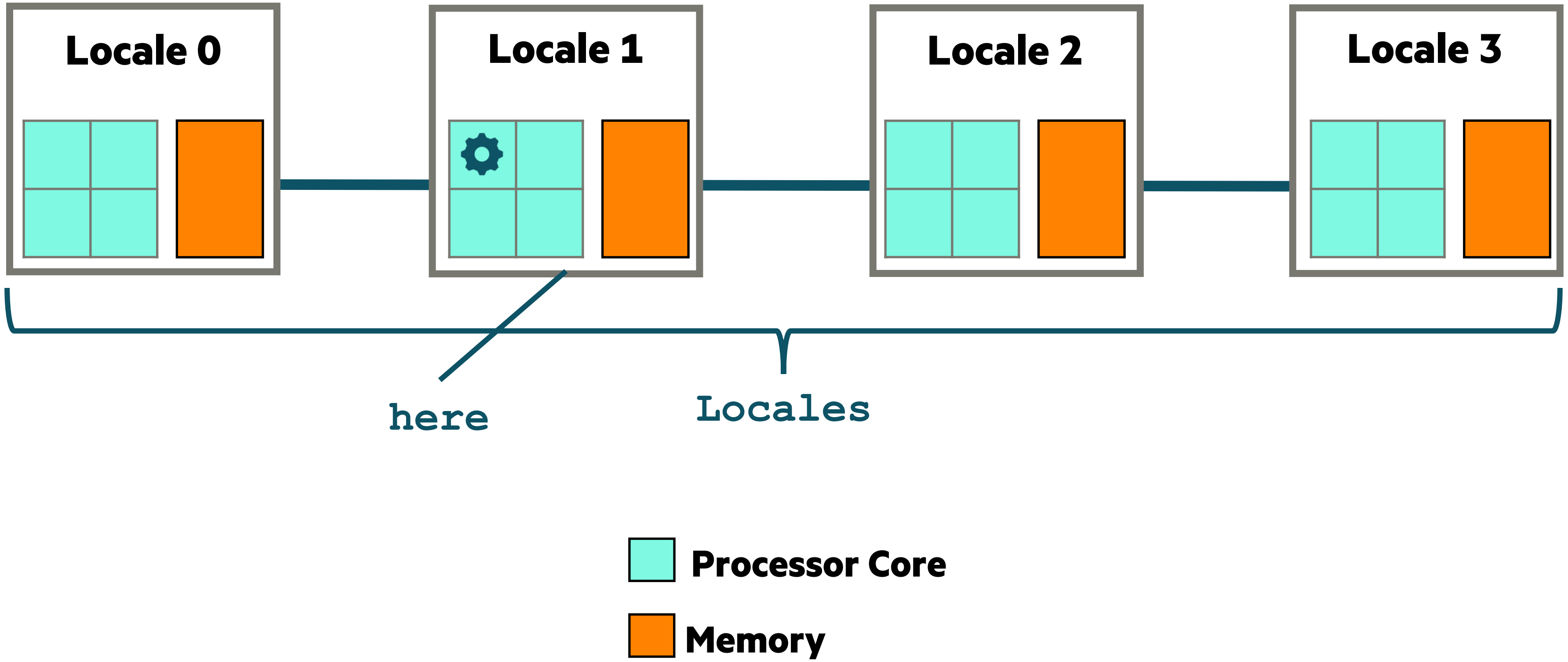


KEY BUILT-IN TYPES AND VARIABLES RELATED TO LOCALES

`locale`: A type that represents system resources on which the program can run

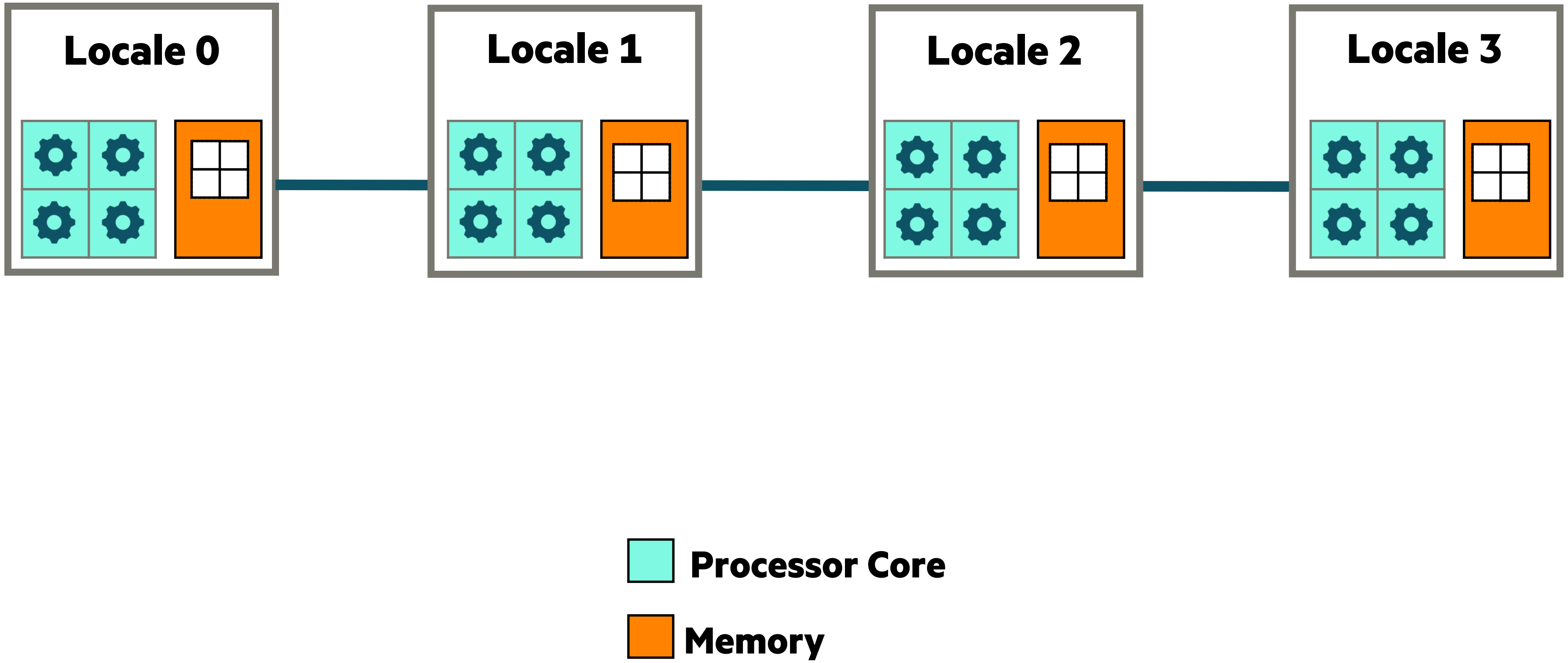
`Locales`: An array of `locale` values

`here` : The `locale` on which the current task is executing

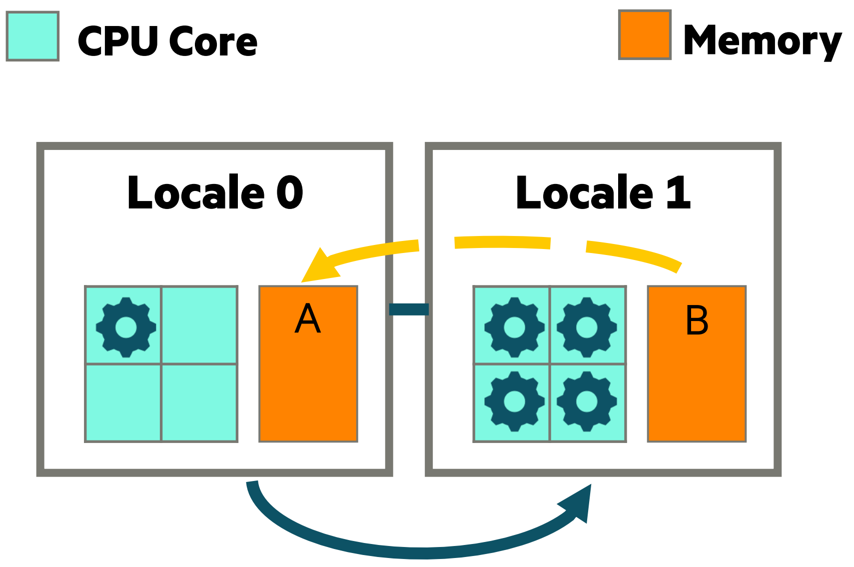


KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

1. **parallelism:** Which tasks should run simultaneously?
2. **locality:** Where should tasks run? Where should data be allocated?



PARALLELISM AND LOCALITY



Execution/allocation moves to Locale 1

```
var A: [1..2, 1..2] real;


  on Locales[1] {
   var B: [1..2, 1..2] real;
   B = 2;
   A = B;
  }




writeln(A);
```

PARALLELISM AND LOCALITY

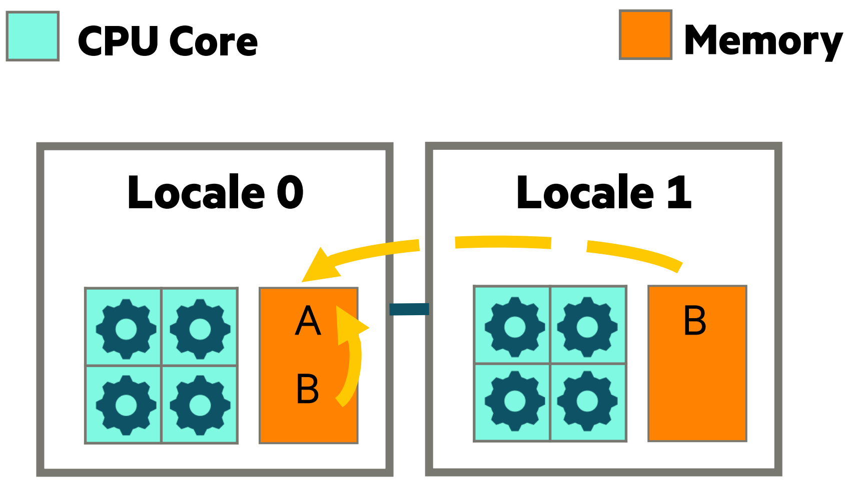
```
forall b in B do
  b = 2;
```

Can be expressed in different ways

```
forall i in B.domain do
  B[i] = 2;
```

```
var A: [1..2, 1..2] real;

  on Locales[1] {
   var B: [1..2, 1..2] real;
   B = 2;
   A = B;
  }

writeln(A);
```

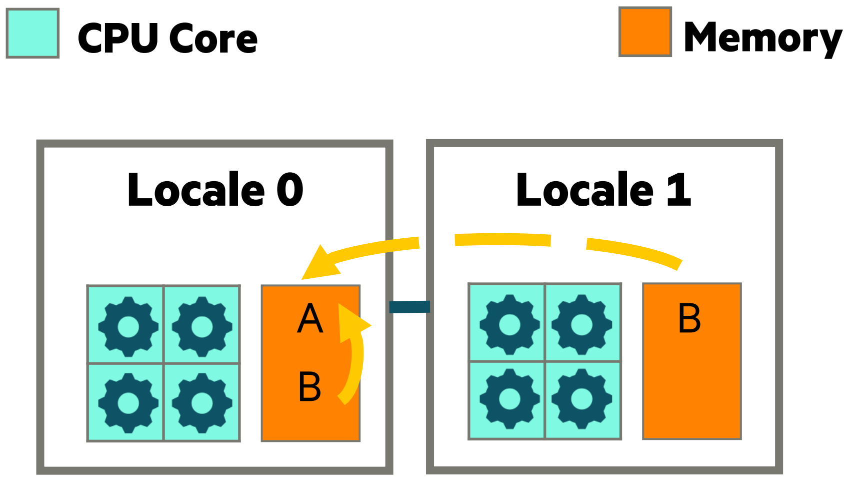
PARALLELISM AND LOCALITY




```
var A: [1..2, 1..2] real;


  for l in Locales do on l {
   var B: [1..2, 1..2] real;
   B = 2;
   A = B;
  }
```

```
writeln(A);
```

PARALLELISM AND LOCALITY

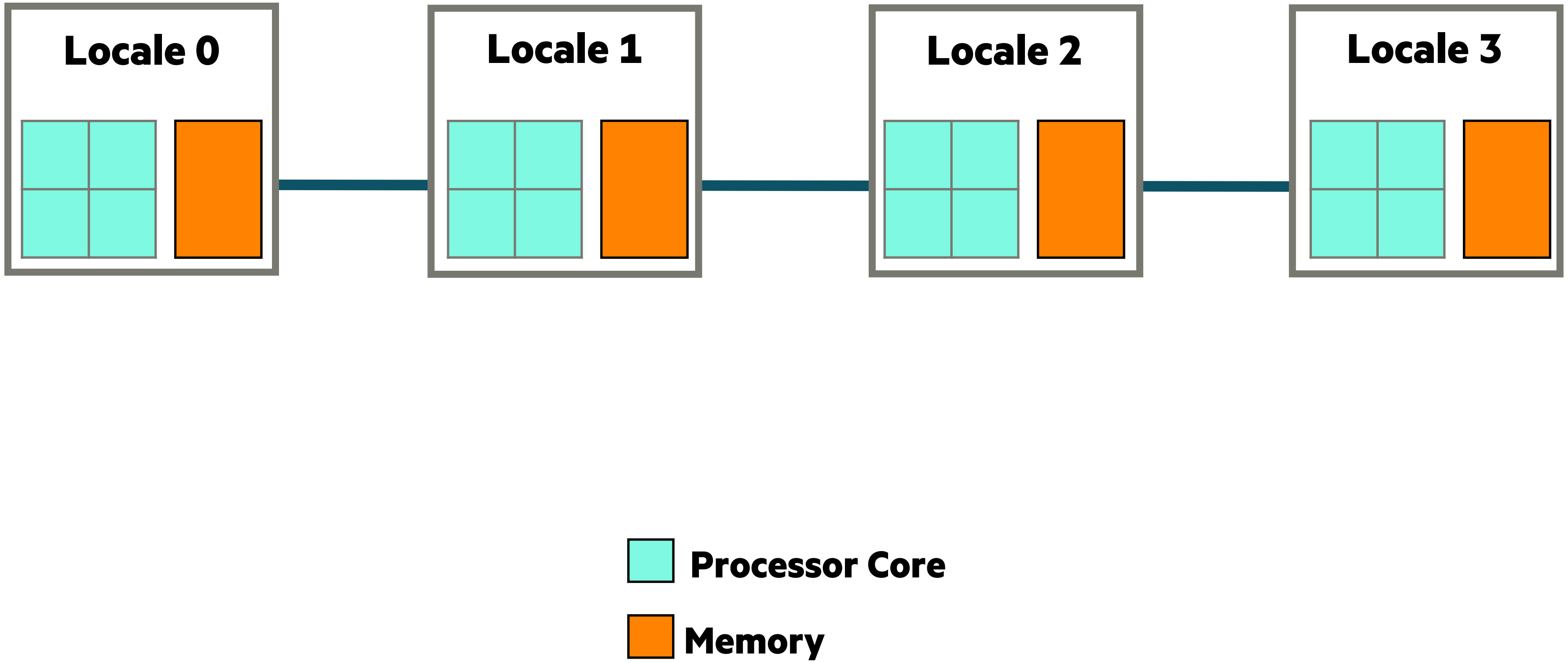


The coforall loop creates
a parallel task per iteration

```
var A: [1..2, 1..2] real;

  coforall l in Locales do on l {
   var B: [1..2, 1..2] real;
   B = 2;
   A = B;
  }

writeln(A);
```

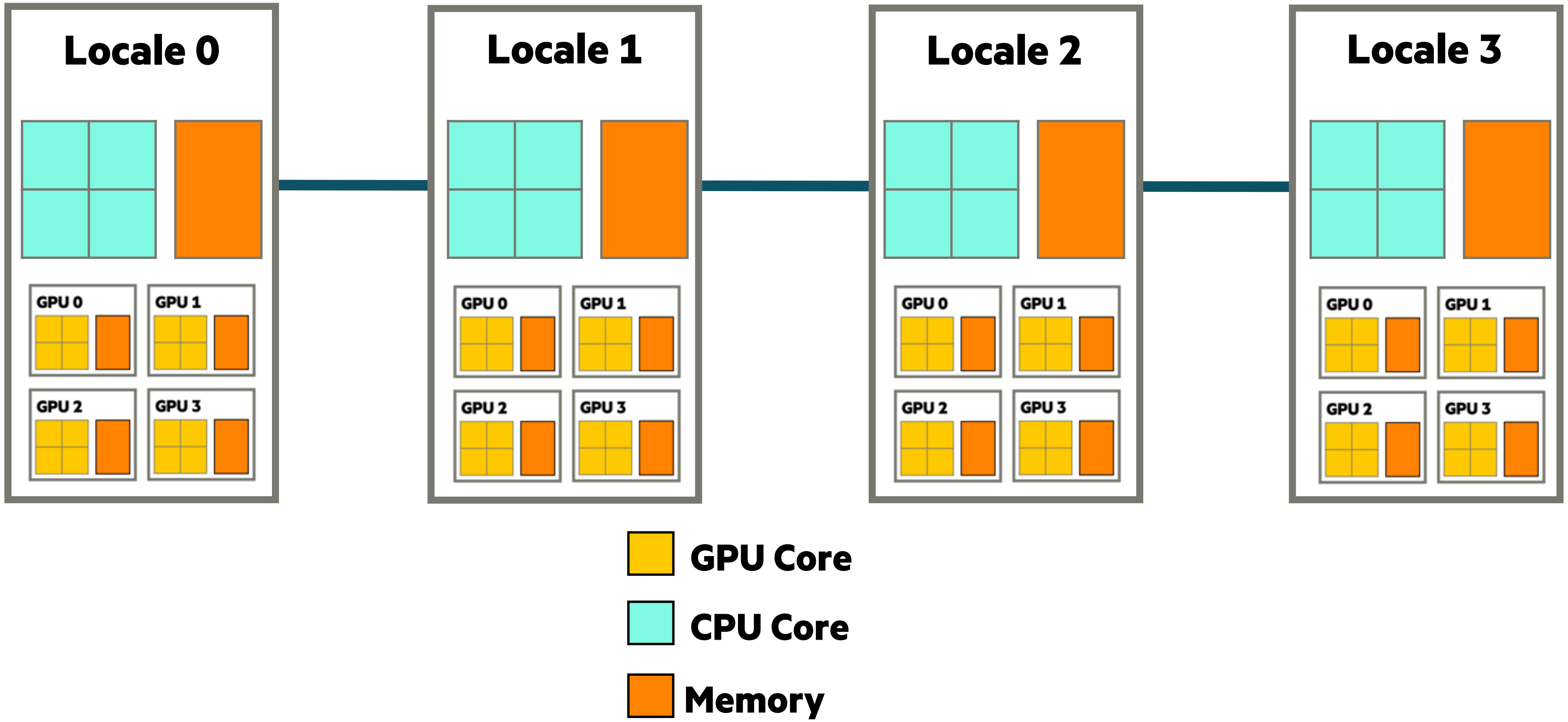
KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

1. **parallelism:** Which tasks should run simultaneously?
2. **locality:** Where should tasks run? Where should data be allocated?
   - complicating matters, compute nodes now often have GPUs with their own processors and memory

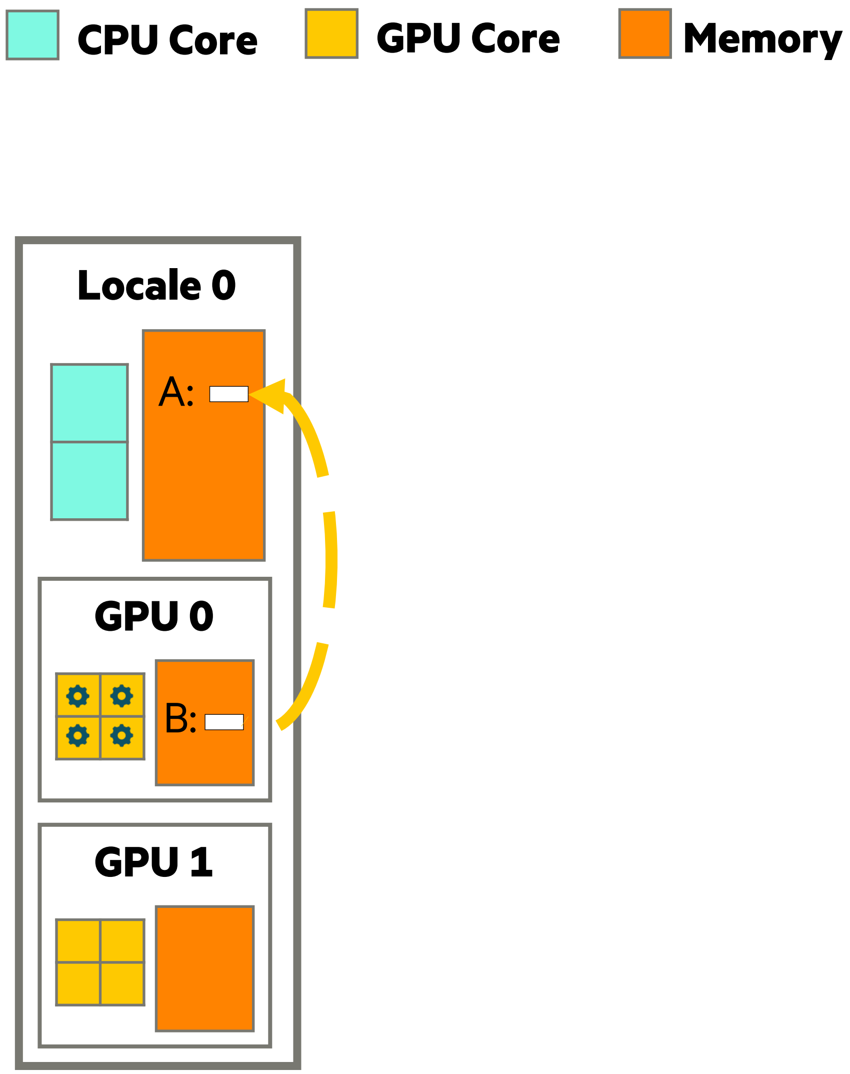


KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

1. **parallelism:** Which tasks should run simultaneously?
2. **locality:** Where should tasks run? Where should data be allocated?
   - complicating matters, compute nodes now often have GPUs with their own processors and memory
   - we represent these as *sub-locales* in Chapel



PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS




```
const nProcs = 1,
      nRows = nProcs,
      nCols = 5;

var A: [1..nRows, 1..nCols] real;


    on here.gpus[0] {
     var B: [1..nRows, 1..nCols] real;
     B = 2;
     A = B;
    }


writeln(A);
```

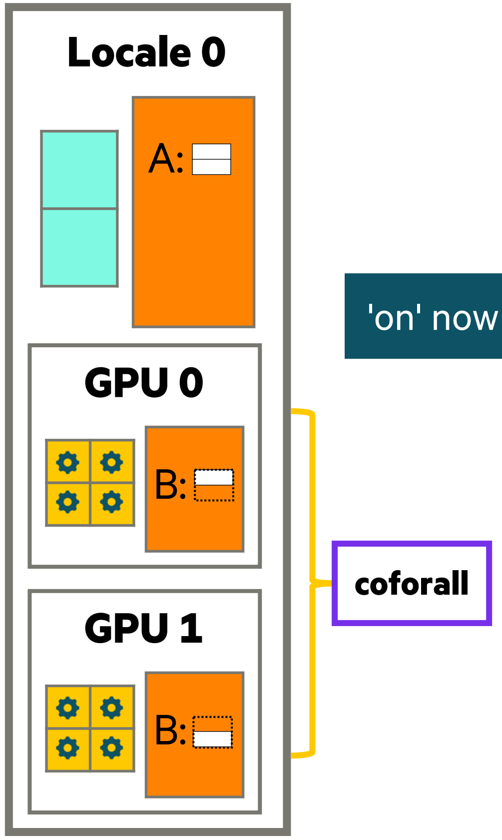
We'll work with a single GPU in this step

Number of rows and columns in the array

2D array of real values

PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS

CPU Core GPU Core Memory




```
const nProcs = here.gpus.size,
      nRows = nProcs,
      nCols = 5;

var A: [1..nRows, 1..nCols] real;

coforall (gpu, gRow) in zip(here.gpus, 1..) {
  on gpu {
   var B: [1..nCols] real;
   B = 2;
   A[gRow, ..] = B;
  }
}

writeln(A);
```

Now, we'll use all the local GPUs

Each iteration of 'coforall' will run in parallel

Iterating GPUs and 1.. in lockstep manner

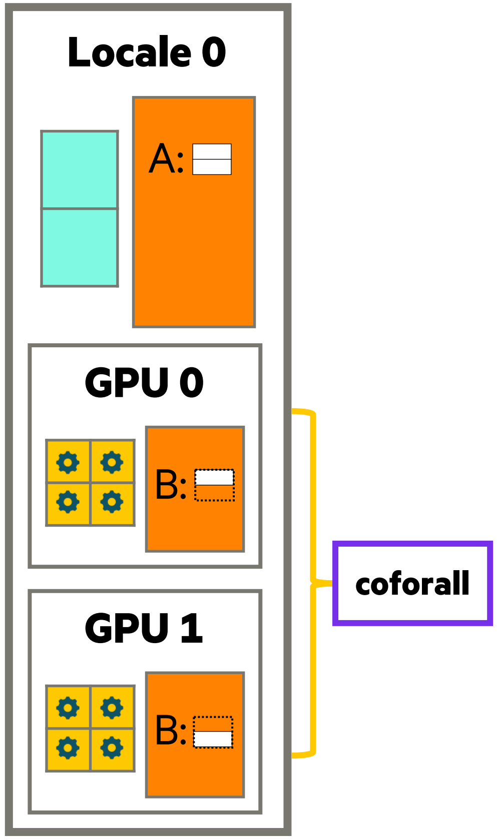
'on' now targets the yielded GPU

Per-GPU array is now 1D

Assigning B to only a single row of A

PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS

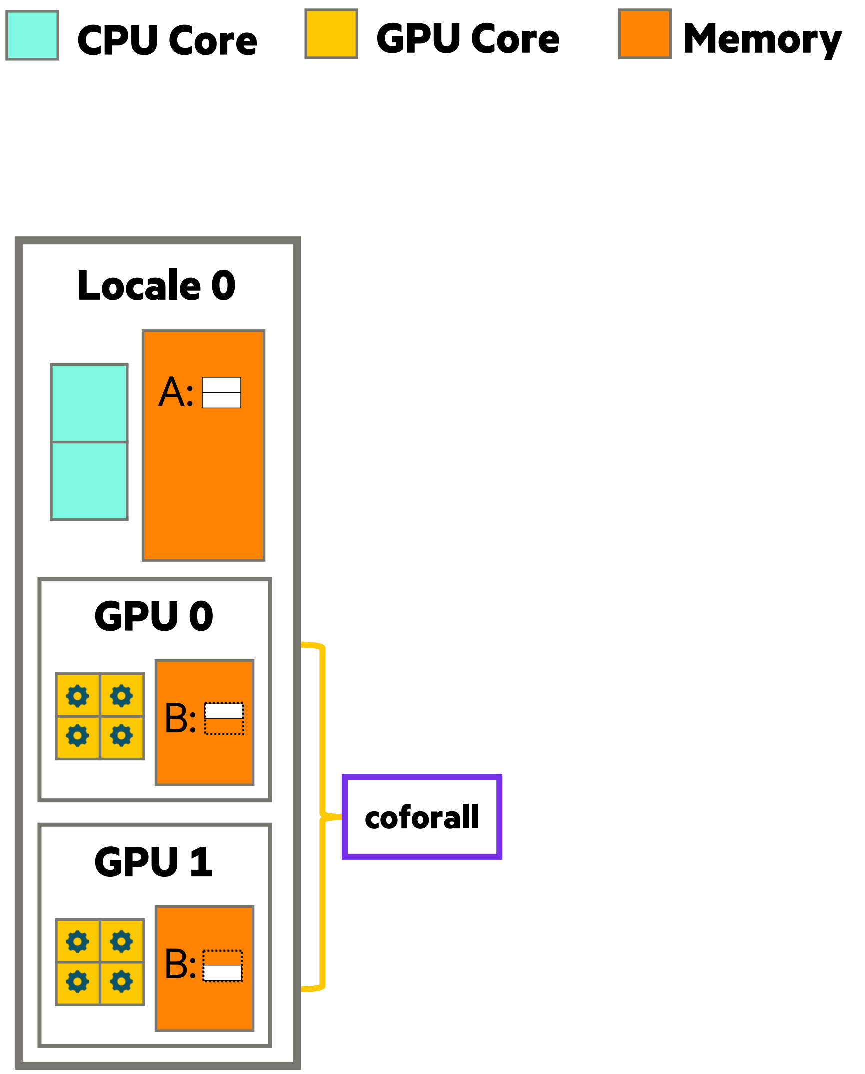
CPU Core · GPU Core · Memory




```
const nProcs = here.gpus.size,
      nRows = nProcs,
      nCols = 5;

var A: [1..nRows, 1..nCols] real;



   coforall (gpu, gRow) in zip(here.gpus, 1..) {
    on gpu do setRowToTwo(gRow);

 }
}
writeln(A);

proc setRowToTwo(row) {
 var B: [1..nCols] real;
 B = 2;
 A[row, ..] = B;
}
```

Small refactor to put the application logic in a function

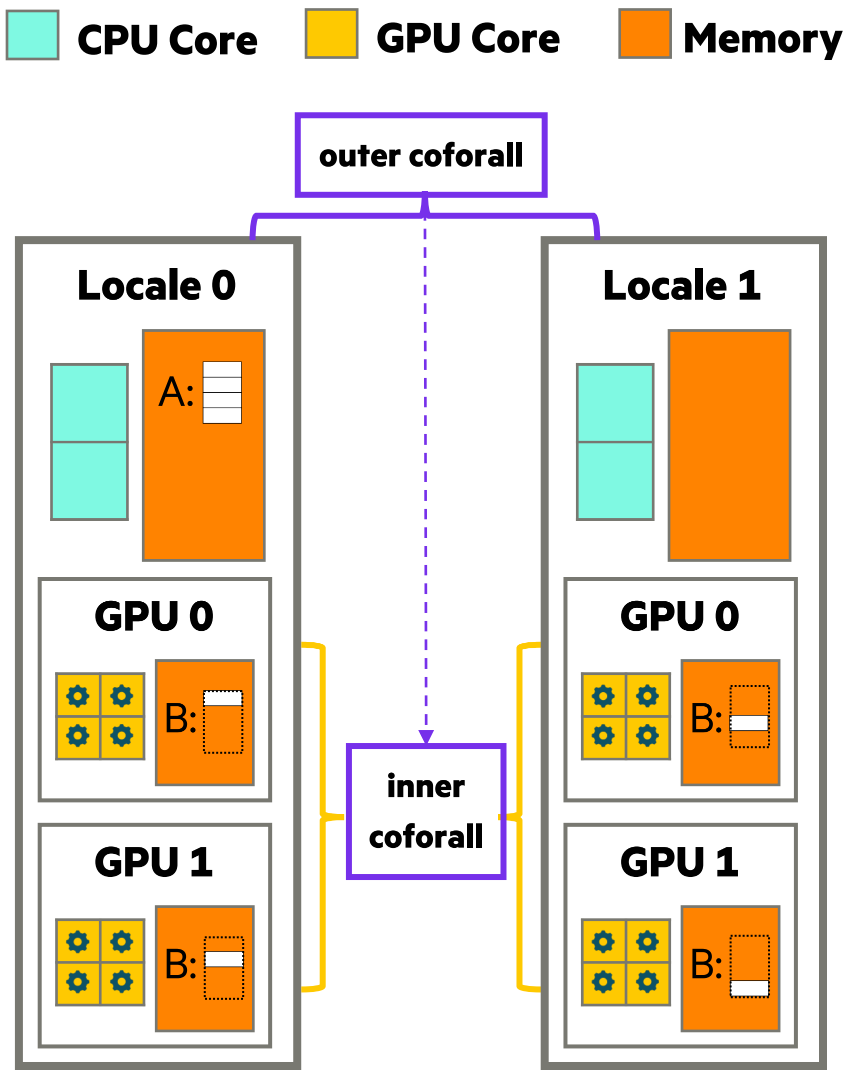
PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS




```
const nProcs = here.gpus.size,
      nRows = nProcs,
      nCols = 5;

var A: [1..nRows, 1..nCols] real;

   coforall (gpu, gRow) in zip(here.gpus, 1..) {
     on gpu do setRowToTwo(gRow);
   }

writeln(A);

proc setRowToTwo(row)  { /* body omitted */ }
```

Body of this function will always be the same

PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS



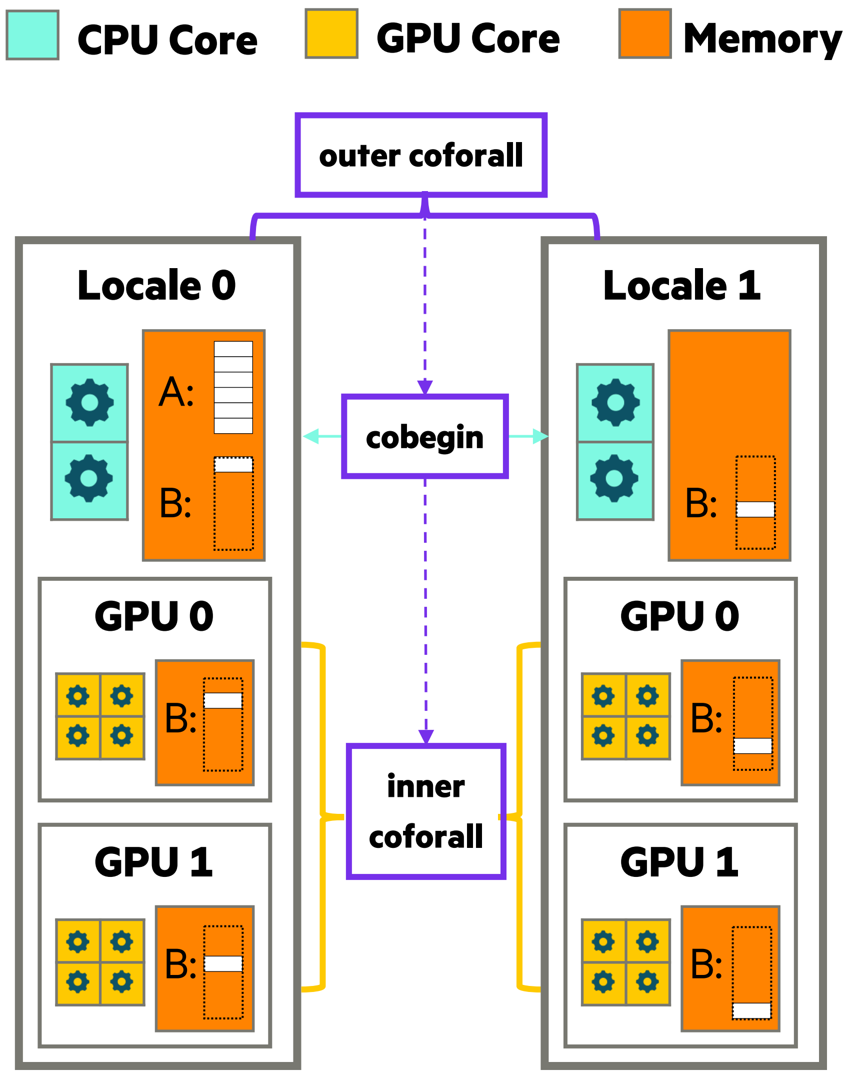

```
const nProcs = here.gpus.size,
      nRows = Locales.size*nProcs,
      nCols = 5;

var A: [1..nRows, 1..nCols] real;

coforall (loc, cRow) in zip(Locales, 1.. by nProcs) {
 on loc {


   coforall (gpu, gRow) in zip(here.gpus, cRow..) {
    on gpu do setRowToTwo(gRow);
   }

 }
}
writeln(A);


proc setRowToTwo(row)  { /* body omitted */ }
```

Now, we also use all locales

Iterate locales and starting row of each locale in a lockstep manner

For each locales' GPUs, we now start the offset at cRow instead of 1

PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS



+1 for the CPU per locale

The 2 statements in 'cobegin' will run in parallel

```
const nProcs = here.gpus.size+1,
      nRows = Locales.size*nProcs,
      nCols = 5;

var A: [1..nRows, 1..nCols] real;

coforall (loc, cRow) in zip(Locales, 1.. by nProcs) {
 on loc {
  cobegin {
1  setRowToTwo(cRow);
   coforall (gpu, gRow) in zip(here.gpus, cRow+1..) {
2   on gpu do setRowToTwo(gRow);
   }
  }
 }
}
writeln(A);


proc setRowToTwo(row) { /* body omitted */ }
```

DIFFERENT TYPES OF PARALLELISM EXPRESSED CONCISELY

```
const nProcs = here.gpus.size+1,
      nRows = Locales.size*nProcs,
      nCols = 5;
var A: [1..nRows, 1..nCols] real;
coforall (loc, cRow) in zip(Locales, 1.. by nProcs) do on loc {
 cobegin {
  setRowToTwo(cRow);
  coforall (gpu, gRow) in zip(here.gpus, cRow+1..) do on gpu {
   setRowToTwo(gRow);
  }
 }
}
writeln(A);

proc setRowToTwo(row) {
 var B: [1..nCols] real;
 B = 2;
 A[row, ..] = B;
}
```

GPU programming in Chapel doesn't require learning new concepts

The only GPU-specific concept in the language
is 'gpus' array on 'locale' type

Full code in a single slide that will use

✓ all nodes
✓ all CPU cores
✓ all GPU cores

Made possible by Chapel's
parallelism and locality constructs

STORIES FROM THE CHAPEL COMMUNITY

CHAPEL PERFORMANCE ON DIFFERENT GPU AND CPUS

- Comparing Chapel's performance
  …against OpenMP, Kokkos, CUDA and HIP
  …on different GPU and CPUs
  …using BabelStream, miniBUDE and TeaLeaf

- Recently presented at
  - Heterogeneity in Computing Workshop (HCW)
  - In conjunction with IPDPS

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**

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

Paper is available at milthorpe.org/wp-content/uploads/2024/03/Milthorpe_HCW2024.pdf

MINIBUDE

- Proxy for BUDE (a protein docking simulation)
  - The computation is very arithmetically intensive and makes significant use of trigonometric functions

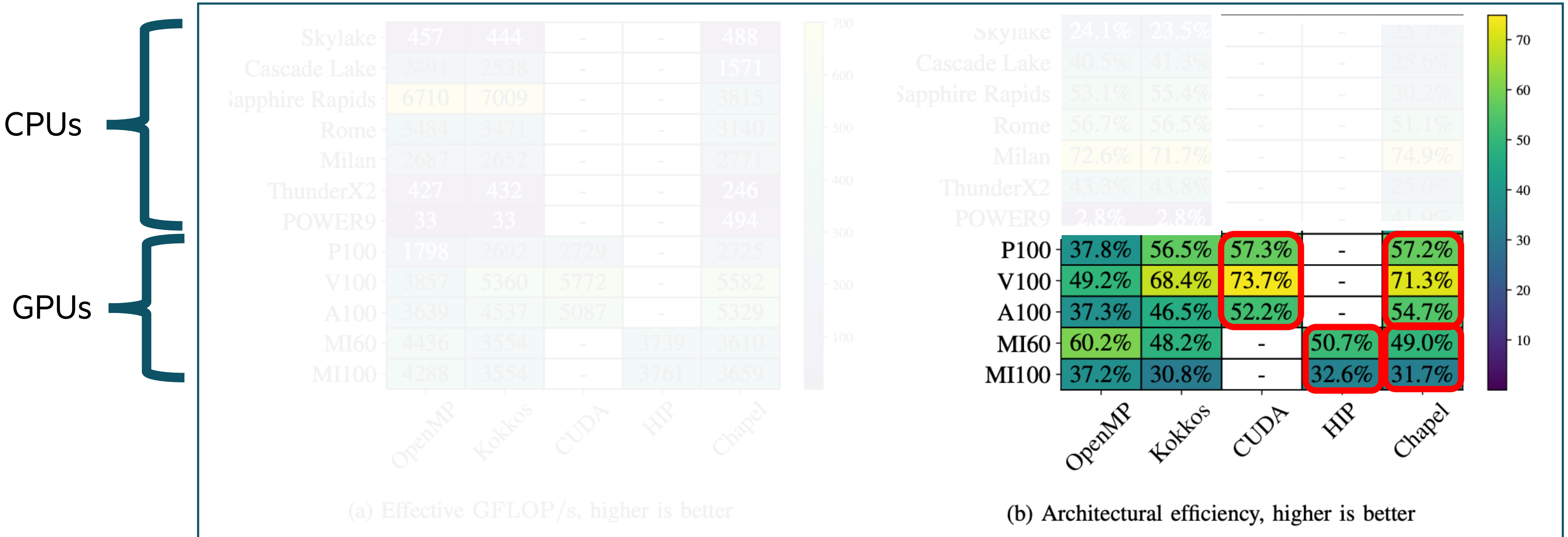
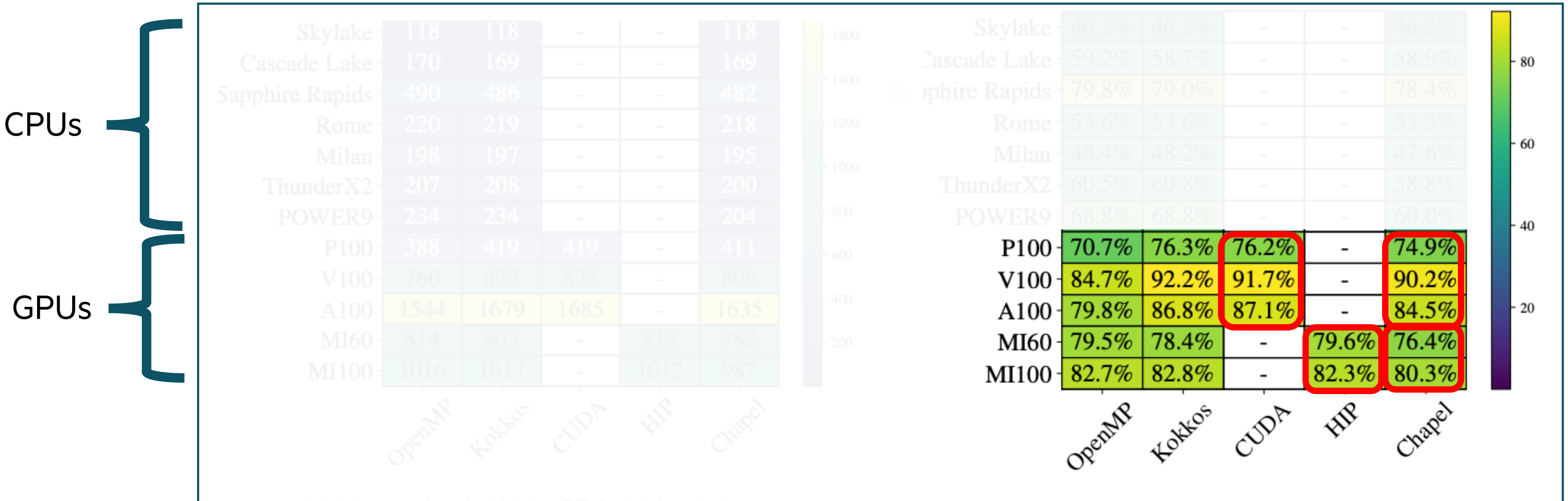


Fig. 2: miniBUDE results for small deck bm1

Figure from: "Performance Portability of the Chapel Language on Heterogeneous Architectures". Josh Milthorpe (Oak Ridge National Laboratory, Australian National University), Xianghao Wang (Australian National University), Ahmad Azizi (Australian National University) Heterogeneity in Computing Workshop (**HCW**)

BABELSTREAM

- Performs stream triad computation computing A = B + α ∗C for arrays A, B, C and scalar α



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

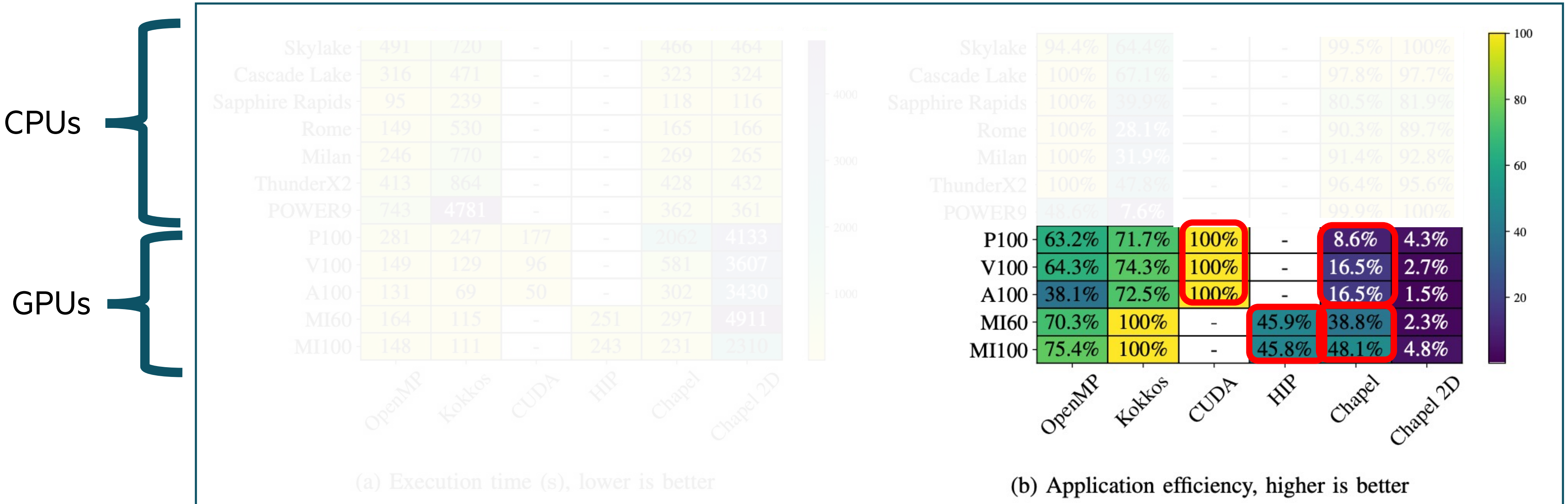
(b) Architectural efficiency, higher is better

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

Figure from: "Performance Portability of the Chapel Language on Heterogeneous Architectures". Josh Milthorpe (Oak Ridge National Laboratory, Australian National University), Xianghao Wang (Australian National University), Ahmad Azizi (Australian National University) Heterogeneity in Computing Workshop (**HCW**)

TEALEAF

- Tealeaf simulates heat conduction over time
- On this application Chapel performed well on CPUs but not GPUs
  - We have some leads for performance issues and still investigating



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

(b) Application efficiency, higher is better

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

Figure from: "Performance Portability of the Chapel Language on Heterogeneous Architectures". Josh Milthorpe (Oak Ridge National Laboratory, Australian National University), Xianghao Wang (Australian National University), Ahmad Azizi (Australian National University). Heterogeneity in Computing Workshop 2024 (**HCW**)

NATIVE GPU PROGRAMMING IN CHAPEL AT SCALE

- Comparing Chapel's native GPU programming
  …against interoperability with HIP and CUDA
  …on Frontier and Perlmutter
  …using N-Queens as proxy for combinatorial optimization

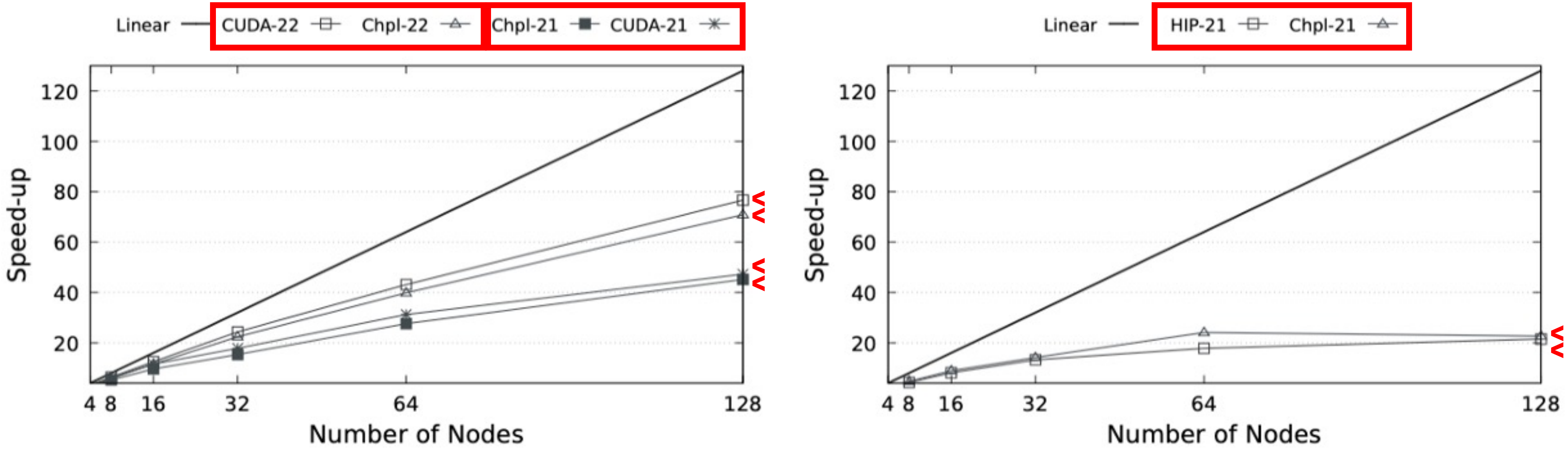
- To be presented at Euro-Par 2024
  - 26-30 August
  - Madrid, Spain

Investigating Portability in Chapel for Tree-based Optimization on GPU-powered Clusters

Tiago Carneiro[1] [0000−0002−6145−8352], Engin Kayraklioglu[2] [0000−0002−4966−3812], Guillaume Helbecque[3,4] [0000−0002−8697−3721], and Nouredine Melab[4]

[1] Interuniversity Microelectronics Centre (IMEC), Belgium
tiago.carneiropessoa@imec.be
[2] Hewlett Packard Enterprise, USA
engin@hpe.com
[3] University of Luxembourg, Luxembourg
guillaume.helbecque@uni.lu
[4] Université de Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, France
nouredine.melab@univ-lille.fr

NATIVE GPU PROGRAMMING IN CHAPEL AT SCALE



(a) NVIDIA-based System

(b) AMD-based system

Figure from: "Investigating Portability in Chapel for Tree-Based Optimizations on GPU-powered Clusters". Tiago Carneiro, Engin Kayraklioglu, Guillaume Helbecque, Nouredine Melab Europar 2024

UPCOMING: KEYNOTE AT CHAPELCON '24

A Case for Parallel-First Languages in Post-Serial, Accelerated World

Paul Sathre, Virginia Tech
June 7th, 2024

Parallel processors have finally dominated all scales of computing hardware, from the personal and portable to the ivory tower datacenters of yore. However, dominant programming models and pedagogy haven't kept pace, and languish in a post-serial mix of libraries and language extensions. Further, heterogeneity in the form of GPUs has dominated the performance landscape of the last decade, penetrating casual user markets thanks to data science, crypto and AI booms. Unfortunately, *GPUs' performance remains largely constrained to expert users by the lack of more productive and portable programming abstractions. This talk will probe questions about how to rethink and democratize parallel programming for the masses*. By reflecting on lessons learned from a decade and a half of accelerated computing, I'll show where Chapel 2.0 fits into the lineage of GPU computing, can capitalize on GPU momentum, and lead a path forward.

ChapelCon '24 is free and fully virtual

chapel-lang.org/ChapelCon24.html

SUMMARY

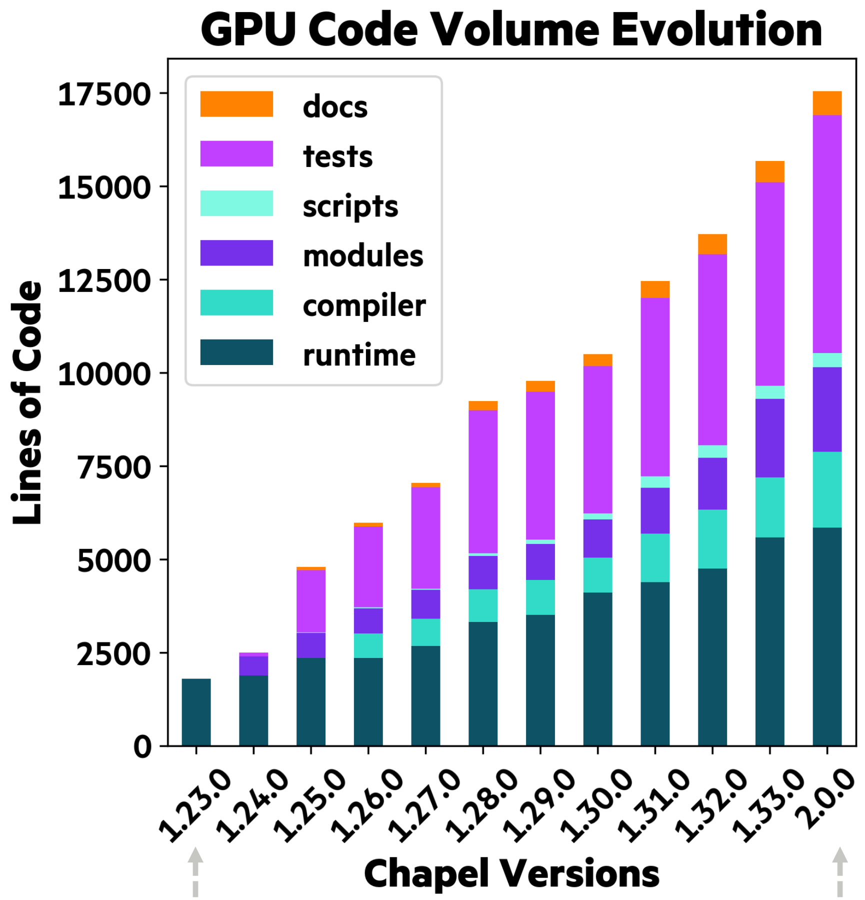
WHERE WE ARE TODAY

Over ~3 years we have been steadily improving

- NVIDIA, AMD GPUs are supported
- Multiple nodes with multiple GPUs can be used
- Parallel tasks can use GPUs concurrently
- GPU features can be emulated on CPUs

Mature enough to get started, big efforts are still underway

- Distributed arrays
- Intel support
- Improving language features to support GPU programming
- Performance improvements
- Bug fixes



October
2020

March
2024

IF YOU WANT TO LEARN MORE ABOUT GPU PROGRAMMING IN CHAPEL

GPU Programming Blog Series: chapel-lang.org/blog/series/gpu-programming-in-chapel/

Introduction to GPU Programming in Chapel

Posted on January 10, 2024.

Tags: GPU Programming How-To

By: Daniel Fedorin

Chapel's High-Level Support for CPU-GPU Data Transfers and Multi-GPU Programming

Posted on April 25, 2024.

Tags: GPU Programming How-To

By: Engin Kayraklioglu

Technote: https://chapel-lang.org/docs/main/technotes/gpu.html

- Anything and everything about our GPU support
  - configuration, advanced features, links to some tests, caveats/limitations
- More of a reference manual than a tutorial

Previous talks

- **LinuxCon / Open Source Summit North America 2024 Talk:** GPU Programming in Chapel and a Live Demo
  - https://youtu.be/5-jLdKduaJE?si=ezaz5mDORvmTjgRL
- **CHIUW '23 Talk:** updates from May '22-May '23 period
  - https://chapel-lang.org/CHIUW/2023/KayrakliogluSlides.pdf
- **LCPC '22 Talk:** a lot of details on how the Chapel compiler works to create GPU kernels
  - https://chapel-lang.org/presentations/Engin-SIAM-PP22-GPU-static.pdf

HPE DEVELOPER MEETUP

Meetup for "Vendor-Neutral GPU Programming in Chapel"

Jul 31, 2024 08:00 AM PDT (-7 UTC)

Jade Abraham, Engin Kayraklioglu

speakers will discuss Chapel's GPU support in detail and collaborate with you to determine how it may help in your particular situation.

HPE developer meetups home page:

https://developer.hpe.com/campaign/meetups/

Registration:

https://hpe.zoom.us/webinar/register/3117139444656/WN_ojVy9LR_QHSCGxeg21rj7A

CHAPELCON '24 (FORMERLY CHIUW, THE CHAPEL IMPLEMENTERS AND USERS WORKSHOP)

Fully virtual, free registration

Schedule Overview

- June 5th: **Tutorial Day**: Chapel and Arkouda tutorials
- June 6th: **Coding Day**: Opportunities for coding
- June 7th: **Conference Day**

Keynote

A Case for Parallel-First Languages in a Post-Serial, Accelerated World

Paul Sathre (*Virginia Tech*)

Registration:

https://chapel-lang.org/ChapelCon24.html

https://chapel-lang.org/blog/posts/chapelcon24/

CHAPEL RESOURCES

Chapel homepage: https://chapel-lang.org

- (points to all other resources)

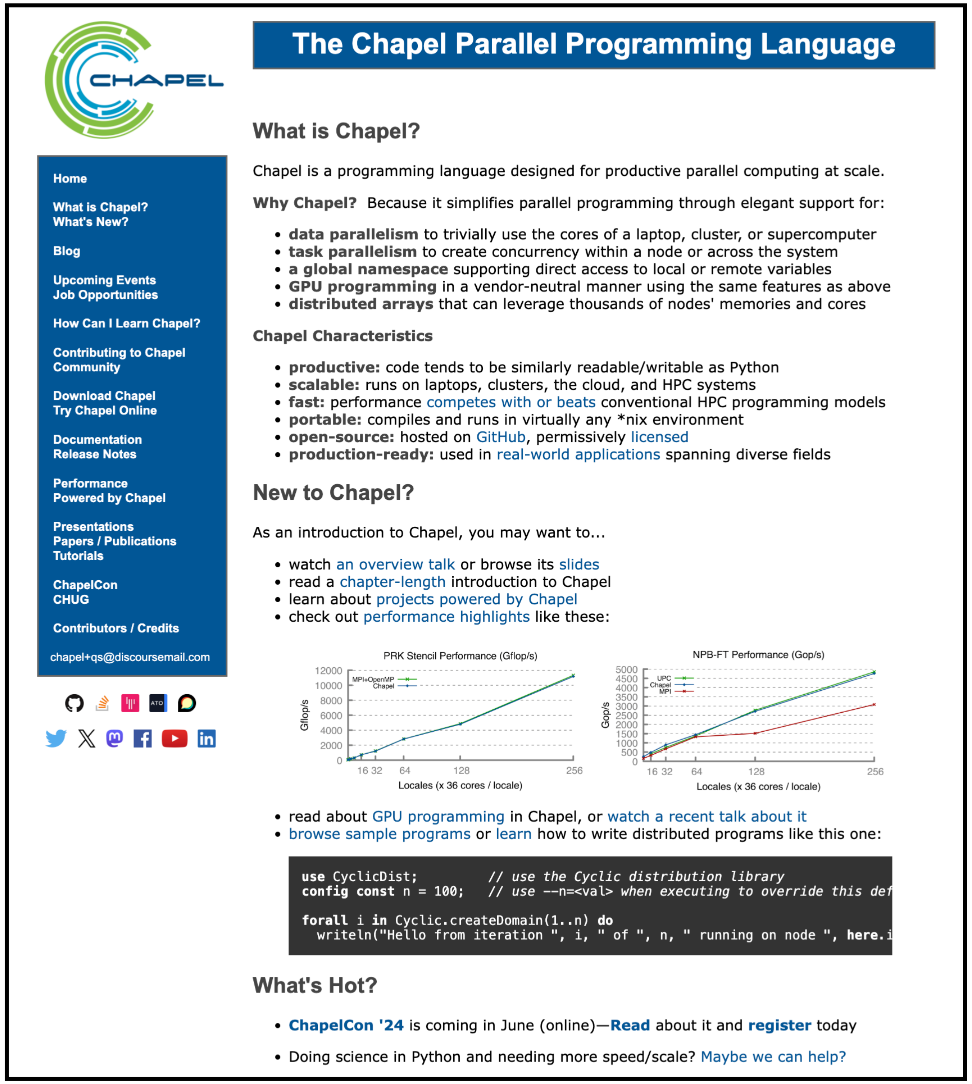
Blog: https://chapel-lang.org/blog/

Social Media:

- Facebook: @ChapelLanguage
- LinkedIn: https://www.linkedin.com/company/chapel-programming-language/
- Mastodon: @ChapelProgrammingLanguage
- X / Twitter: @ChapelLanguage
- YouTube: @ChapelLanguage

Community Discussion / Support:

- Discourse: https://chapel.discourse.group/
- Gitter: https://gitter.im/chapel-lang/chapel
- Stack Overflow: https://stackoverflow.com/questions/tagged/chapel
- GitHub Issues: https://github.com/chapel-lang/chapel/issues



SUMMARY

- GPUs are becoming more and more common in HPC
  - However, programming GPUs is more challenging than programming CPUs
  - On multiple nodes, users are typically required to use multiple paradigms
- HPC and GPUs should be more accessible
  - from wider range of disciplines,
  - with varying levels of expertise, and
  - limited time to invest in programming
- Chapel wants to make HPC more accessible
  - Existing applications prove that Chapel delivers on the promise
  - Its growing support for GPU programming can:
    - enable programming GPUs in a productive and vendor-neutral way
    - provide an all-inclusive solution for programming in HPC
- The Chapel team at HPE would be excited to collaborate with AMD!
  - Please feel free to reach out: engin@hpe.com

chapel-lang.org