Hewlett Packard Enterprise

CUG2024
Diverse Universe
PERTH AUSTRALIA

Portable Support for GPUs and Distributed-Memory Parallelism in Chapel

Andrew Stone, Engin Kayraklioglu

May 9, 2024

It is Hard to Avoid GPUs in HPC

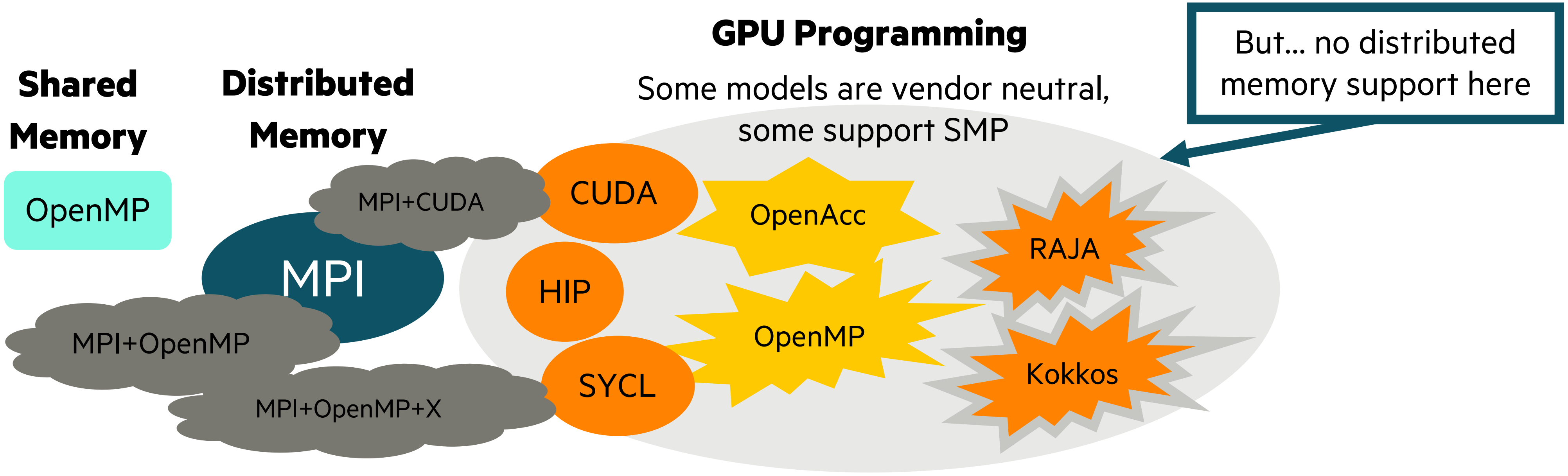
In the top500 list:

- From June 2011 - Nov 2023 there has been a 13x increase in the number of supercomputers with GPUs
- Over the past three years 72% of systems in the top 10 had GPUs

For the #1 system today (Frontier):

- 95% of its compute capability comes from its GPUs

GPUs are Easy to Find… BUT DIFFICULT TO PROGRAM



- 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 less commonly 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

- its goal is to make parallel programming at scale far more productive

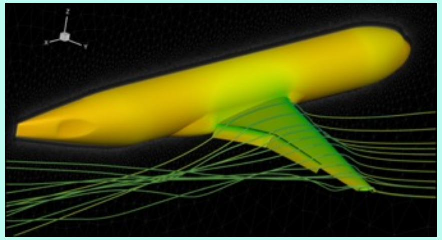
- open-source & collaborative

- portable & scalable
  - works on everything from your laptop to a supercomputer
  - Linux laptops/clusters, Cray systems, MacOS, WSL, AWS, Raspberry Pi
  - shown to scale on Cray networks (Slingshot, Aries), InfiniBand, RDMA-Ethernet
  - NVIDIA and AMD GPUs

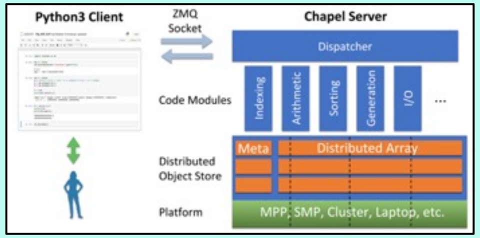
chapel-lang.org

Applications of Chapel

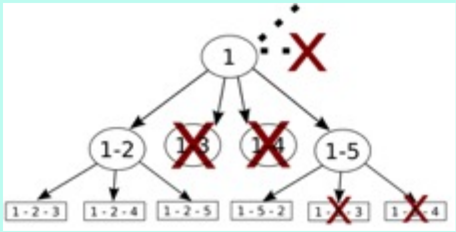
Active GPU efforts



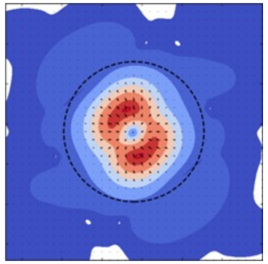
CHAMPS: 3D Unstructured CFD
Laurendeau, Bourgault-Côté, Parenteau, Plante, et al.
École Polytechnique Montréal



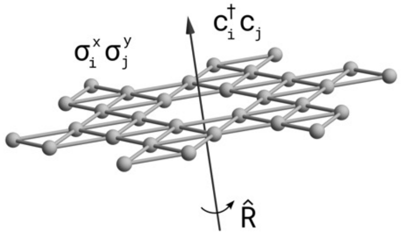
Arkouda: Interactive Data Science at Massive Scale
Mike Merrill, Bill Reus, et al.
U.S. DoD



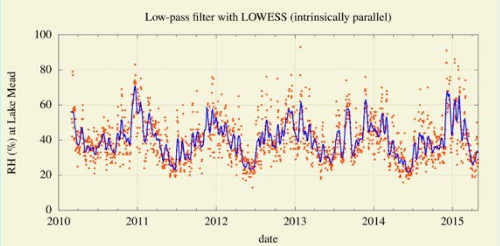
ChOp: Chapel-based Optimization
T. Carneiro, G. Helbecque, N. Melab, et al.
INRIA, IMEC, et al.



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

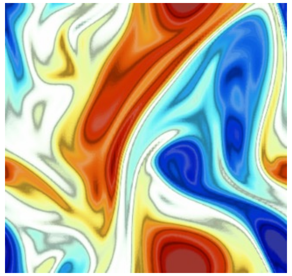


Lattice-Symmetries: a Quantum Many-Body Toolbox
Tom Westerhout
Radboud University

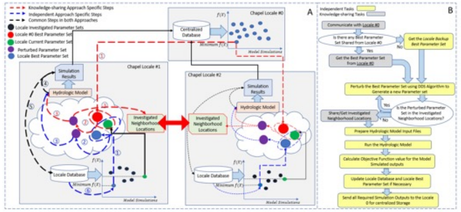


Desk dot chpl: Utilities for Environmental Eng.
Nelson Luis Dias
The Federal University of Paraná, Brazil

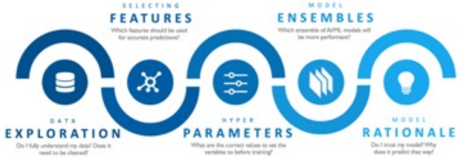
RapidQ: Mapping Coral Biodiversity
Rebecca Green, Helen Fox, Scott Bachman, et al.
The Coral Reef Alliance



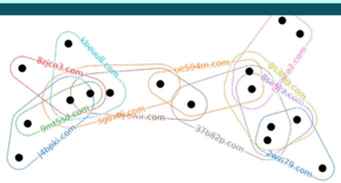
ChapQG: Layered Quasigeostrophic CFD
Ian Grooms and Scott Bachman
University of Colorado, Boulder et al.



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)

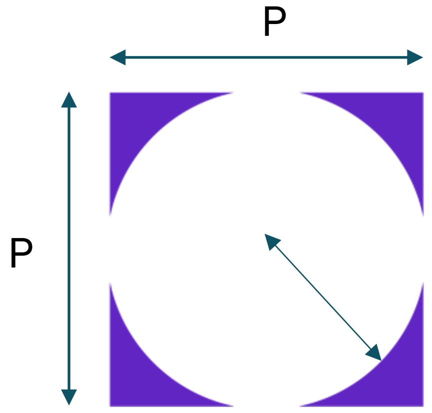
Use Case: Coral Reef Code

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.

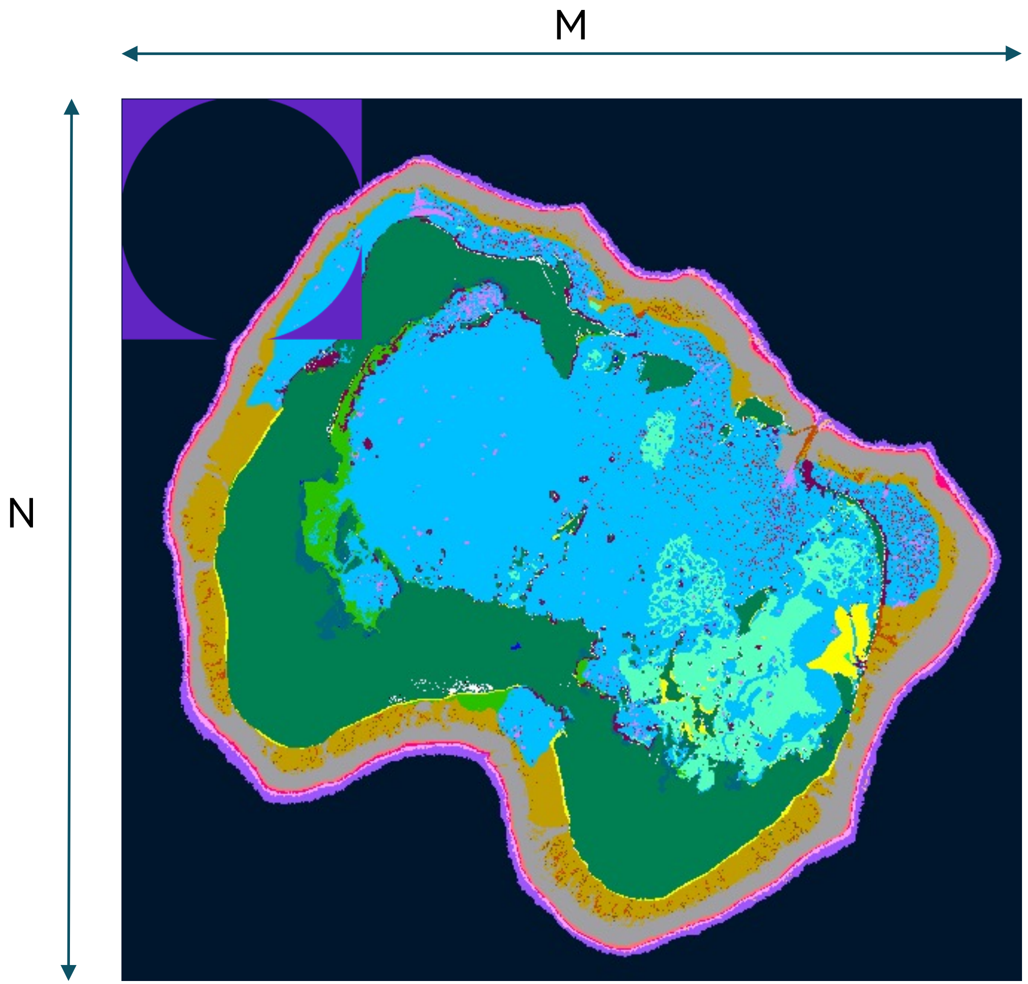


Algorithmic complexity: $O(MNP^3)$

Typically:

- M, N > 10,000

- P ~ 400



For more info see: "High-Performance Programming and Execution of a Coral Biodiversity Mapping Algorithm Using Chapel" by Scott Bachman et al. CHIUW 2023

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...
    var GpuInputArr = InputArr[...];
    var GpuOutputArr: ...;
    convolve(GpuInputArr, GpuOutputArr);
    OutputArr[...] = GpuOutputArr;
}}}
```

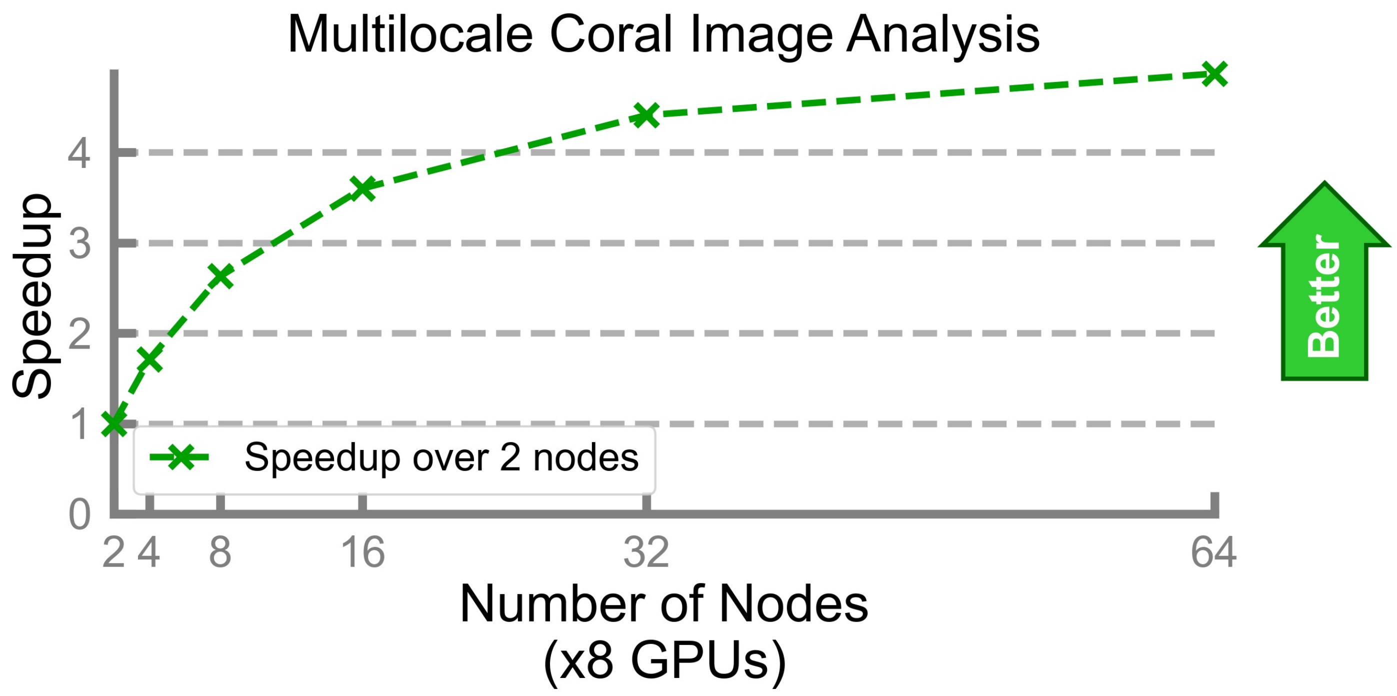
Using a different loop flavor to enable GPU execution.

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

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

Coral Reef Takeaway Points

- Runs on multiple nodes on Frontier!
  - 5x improvement going from 2 to 64 nodes (16 to 512 GPUs)
- Turned sequential Chapel code into multi-node, multi-GPU per node enabled code with minimal changes
- The same code runs on both NVIDIA and AMD GPUs



Looking at performance: MiniBude, BabelStream, TeaLeaf, and ChOp

GPU-enabled Chapel Applications and Performance

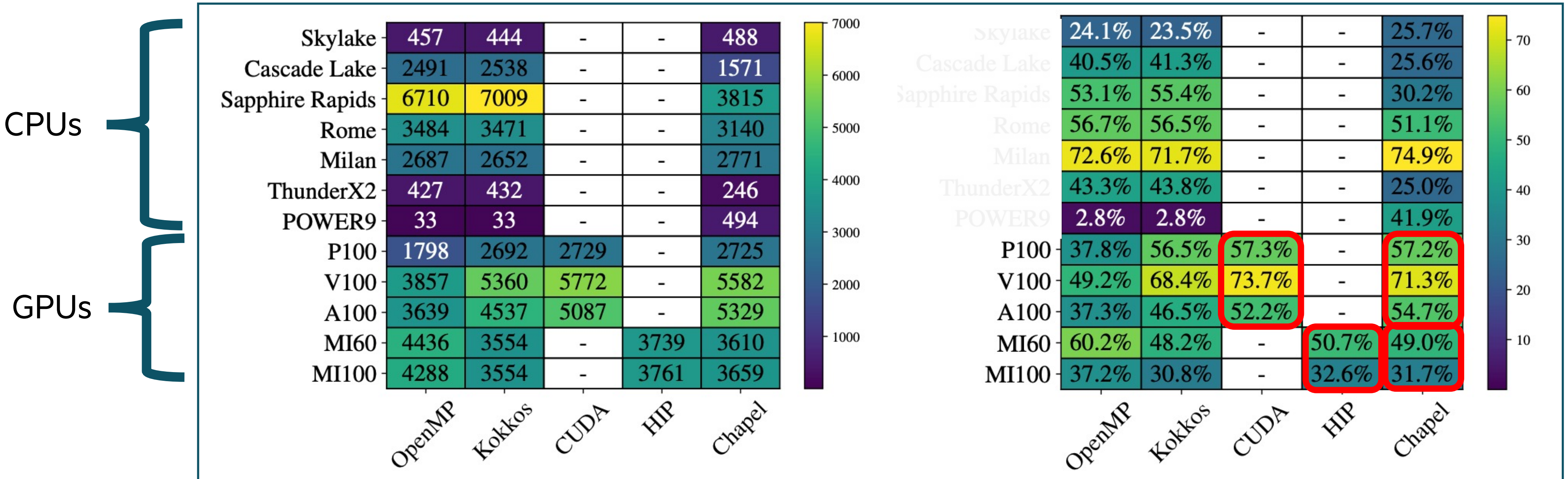
- We discussed Coral Reef application and showed its performance on Frontier
- In the follow slides we give performance results for a few additional miniapps/applications
  - Results were copied directly from the relevant papers (with the authors' permission)
- All these run on both NVIDIA and AMD GPUs and contain no vendor-specific code

- **BabelStream, MiniBude, and TeaLeaf**
  - Chapel implementations by Josh Milthorpe (Oak Ridge National Lab and Australian National University) et al.
  - results are from a paper accepted for the 2024 Heterogeneity in Computing Workshop HCW (part of IPDPS)
    - "Performance Portability of the Chapel Language on Heterogeneous Architectures".

- **ChOp** (Chapel Optimization)
  - written by Tiago Carneiro (Interuniversity Microelectronics Centre (IMEC), Belgium) et al.
  - Solves N-Queens problem
  - results shown are from a submission to EuroPar (currently pending review)

MiniBude: Chapel implementation by Josh Milthorpe from ORNL

- MiniBude is miniapp of Bude (a protein docking simulation)
  - The computation is very arithmetically intensive and makes significant use of trigonometric functions
- For this miniapp, Chapel's performance is close to CUDA's and HIP's
- Architectural efficiency = % of peak memory bandwidth for each platform



(a) Effective GFLOP/s, higher is better

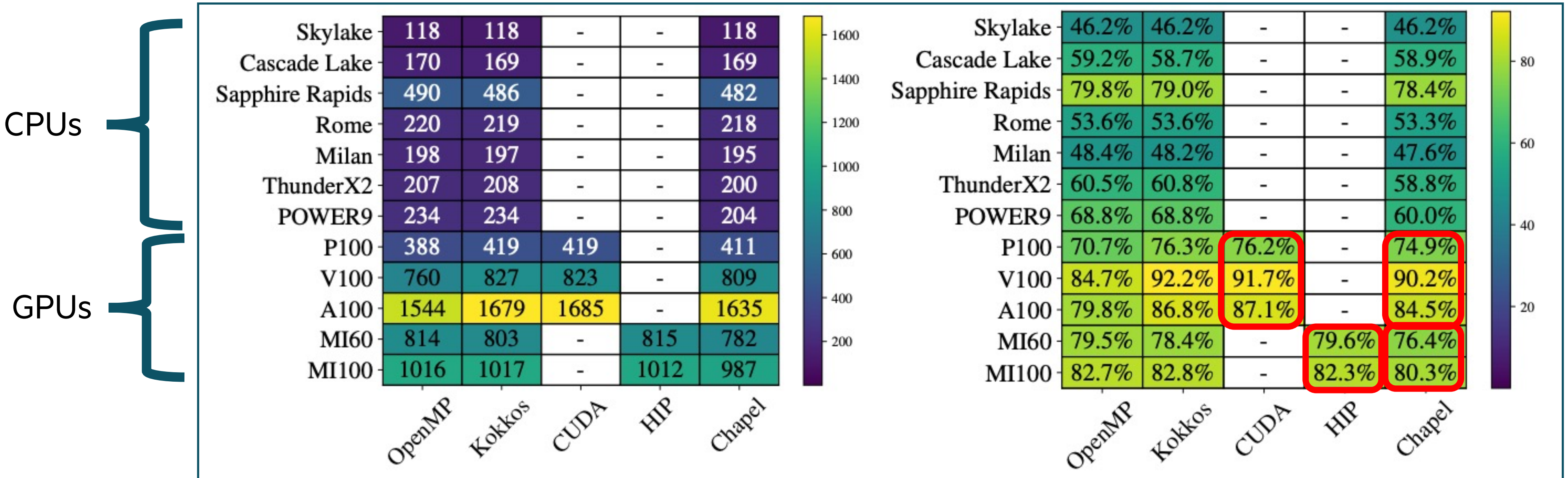
(b) Architectural efficiency, higher is better

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: Chapel implementation by Josh Milthorpe from ORNL

- Performs stream triad computation computing A = B + α ∗C for arrays A, B, C and scalar α
- Chapel performs competitively for this benchmark
- Architectural efficiency = % of peak memory bandwidth for each platform



	OpenMP	Kokkos	CUDA	HIP	Chapel
Skylake	118	118	-	-	118
Cascade Lake	170	169	-	-	169
Sapphire Rapids	490	486	-	-	482
Rome	220	219	-	-	218
Milan	198	197	-	-	195
ThunderX2	207	208	-	-	200
POWER9	234	234	-	-	204
P100	388	419	419	-	411
V100	760	827	823	-	809
A100	1544	1679	1685	-	1635
MI60	814	803	-	815	782
MI100	1016	1017	-	1012	987

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

	OpenMP	Kokkos	CUDA	HIP	Chapel
Skylake	46.2%	46.2%	-	-	46.2%
Cascade Lake	59.2%	58.7%	-	-	58.9%
Sapphire Rapids	79.8%	79.0%	-	-	78.4%
Rome	53.6%	53.6%	-	-	53.3%
Milan	48.4%	48.2%	-	-	47.6%
ThunderX2	60.5%	60.8%	-	-	58.8%
POWER9	68.8%	68.8%	-	-	60.0%
P100	70.7%	76.3%	76.2%	-	74.9%
V100	84.7%	92.2%	91.7%	-	90.2%
A100	79.8%	86.8%	87.1%	-	84.5%
MI60	79.5%	78.4%	-	79.6%	76.4%
MI100	82.7%	82.8%	-	82.3%	80.3%

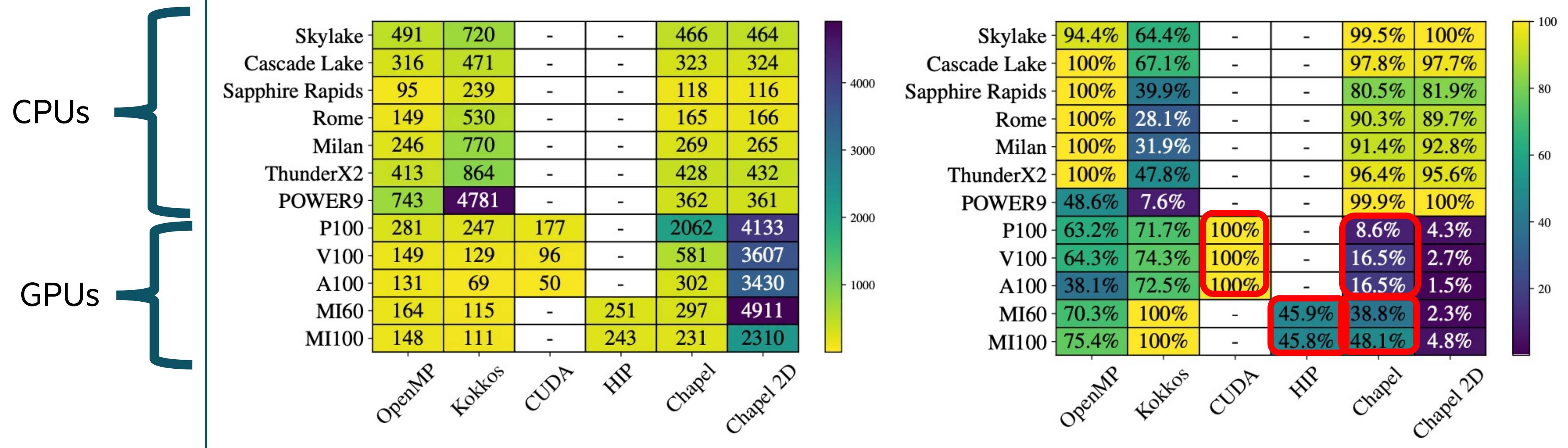
(b) Architectural efficiency, higher is better

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: Chapel implementation by Josh Milthorpe from ORNL

- Tealeaf simulates heat conduction over time
- On this application Chapel performed well on CPUs but not GPUs
  - We are investigating this and suspect better in-kernel reduction support will help close the gap
- Application efficiency = performance relative to fastest implementation for each platform



(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 (**HCW**)

ChOp: N-Queens Solver by Tiago Carneiro from IMEC

- Results are shown for two different problem sizes "21" and "22"
- The "CUDA"/"HIP" versions use Chapel's interoperability features to launch kernels written in CUDA/HIP
- For size=21 Chapel and CUDA/HIP perform similarly well, for size=22 the HIP version would crash so we don't have comparative results for that (the Chapel version would, however, scale)

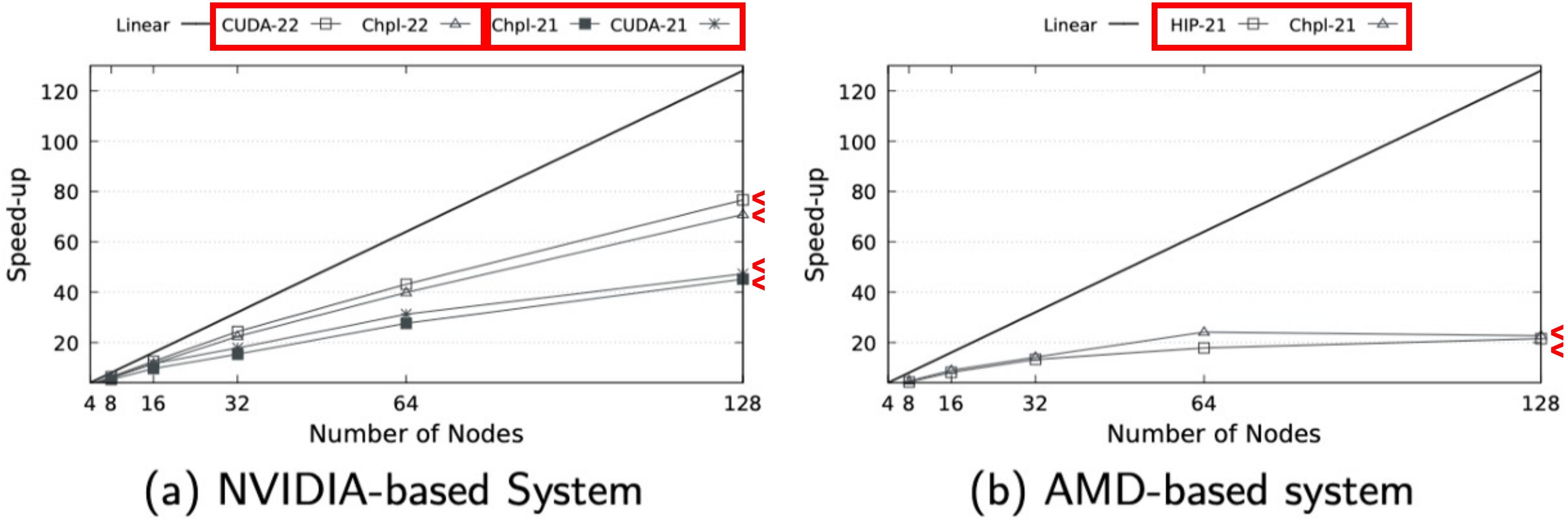
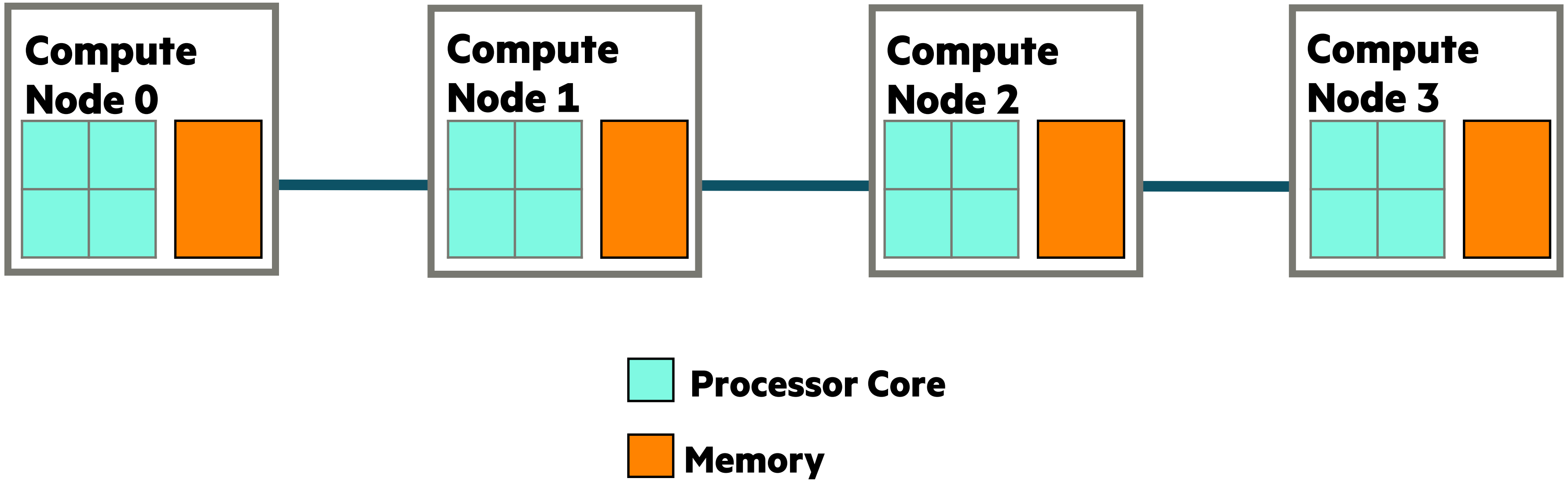


Figure from: "Investigating Portability in Chapel for Tree-Based Optimizations on GPU-powered Clusters". Tiago Carneiro, Engin Kayraklioglu, Guillaume Helbecque, Nouredine Melab

Programming GPUs using GPU locales

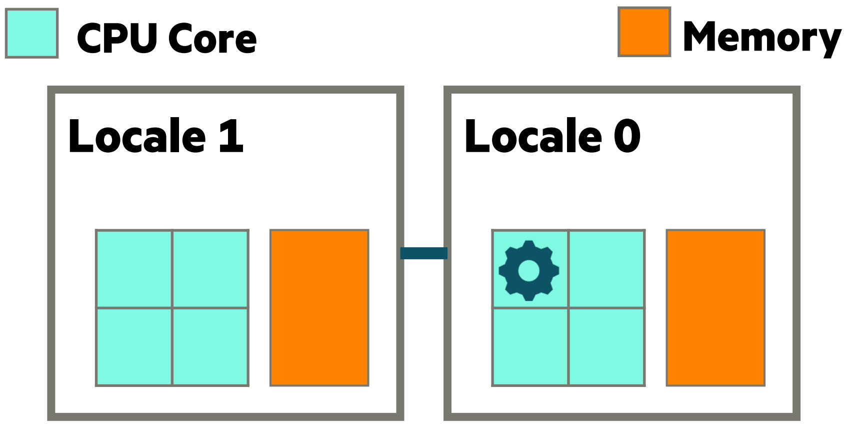
Locales in Chapel

- Locales represent the resources of your HPC system that have:
  - processors, so it can run tasks
  - memory, so it can store variables



Using Locales in Chapel

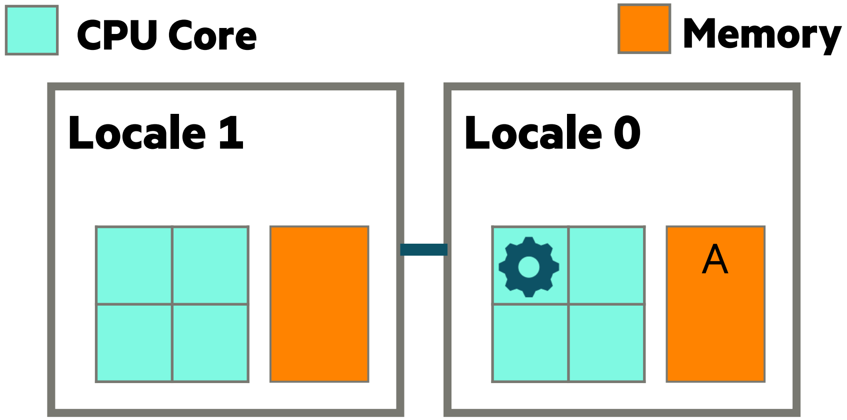
- Execution starts with a single task running on the first locale (i.e. **Locale[0]**)




```
// Execution starts on Locale[0]
var A: [1..4] real;
coforall i in 1..4 do
  A[i] = someComputation(i);
on Locales[1] {
  var B: [1..4] real;
  B = 2;
  A = B;
}
```

Using Locales in Chapel

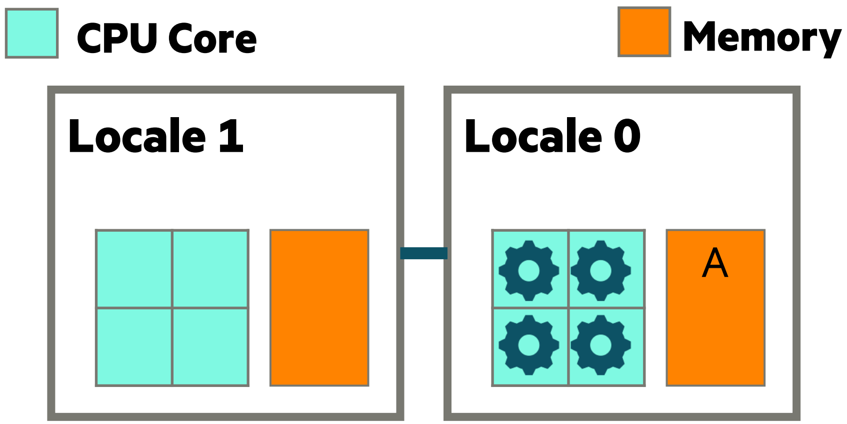
- Execution starts with a single task running on the first locale (i.e. **Locale[0]**)




```
// Execution starts on Locale[0]
var A: [1..4] real;
coforall i in 1..4 do
  A[i] = someComputation(i);
on Locales[1] {
  var B: [1..4] real;
  B = 2;
  A = B;
}
```

Using Locales in Chapel

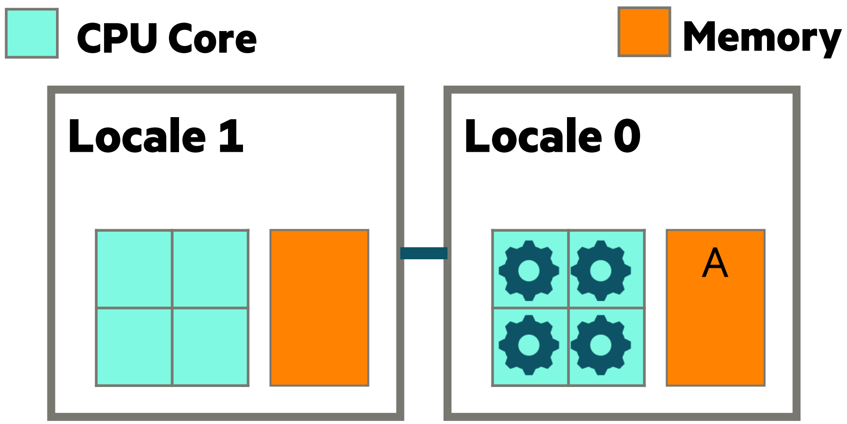
- Execution starts with a single task running on the first locale (i.e. **Locale[0]**)
- Use **coforall** loops to spawn new tasks (one per loop iteration); tasks synchronize at end of loop
- Use **on** statements to move an executing task from one locale to another




```
// Execution starts on Locale[0]
var A: [1..4] real;
coforall i in 1..4 do
  A[i] = someComputation(i);
on Locales[1] {
  var B: [1..4] real;
  B = 2;
  A = B;
}
```

Using Locales in Chapel

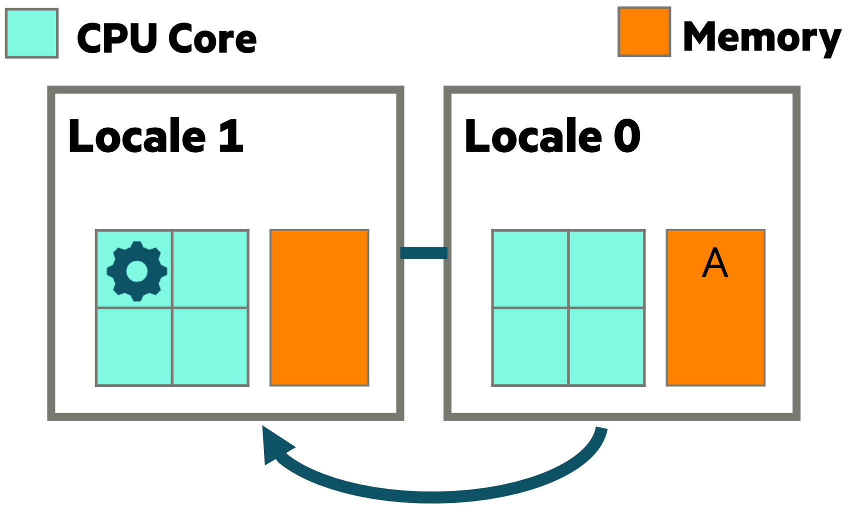
- Execution starts with a single task running on the first locale (i.e. **Locale[0]**)
- Use **coforall** loops to spawn new tasks (one per loop iteration); tasks synchronize at end of loop
- Use **on** statements to move an executing task from one locale to another




```
// Execution starts on Locale[0]
var A: [1..4] real;
coforall i in 1..4 do
  A[i] = someComputation(i);
on Locales[1] {
  var B: [1..4] real;
  B = 2;
  A = B;
}
```

Using Locales in Chapel

- Execution starts with a single task running on the first locale (i.e. **Locale[0]**)
- Use **coforall** loops to spawn new tasks (one per loop iteration); tasks synchronize at end of loop
- Use **on** statements to move an executing task from one locale to another
- The Locales array contains locales for all the nodes in your system

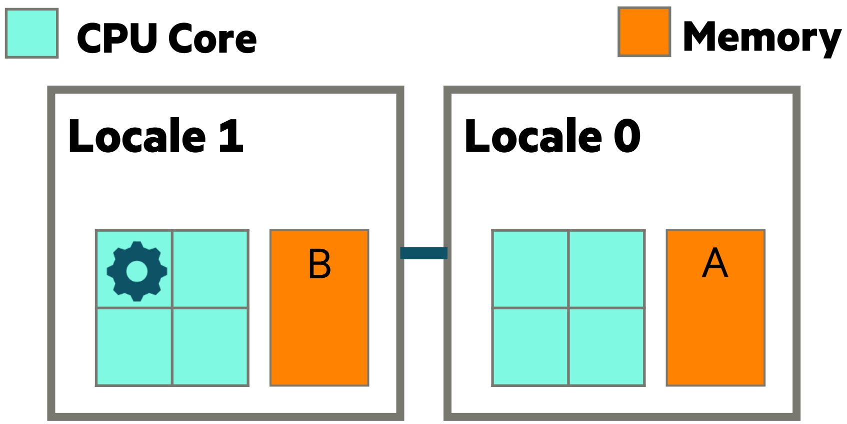


Execution/allocation
moves to Locale 1

```
// Execution starts on Locale[0]
var A: [1..4] real;
coforall i in 1..4 do
  A[i] = someComputation(i);
on Locales[1] {
  var B: [1..4] real;
  B = 2;
  A = B;
}
```

Using Locales in Chapel

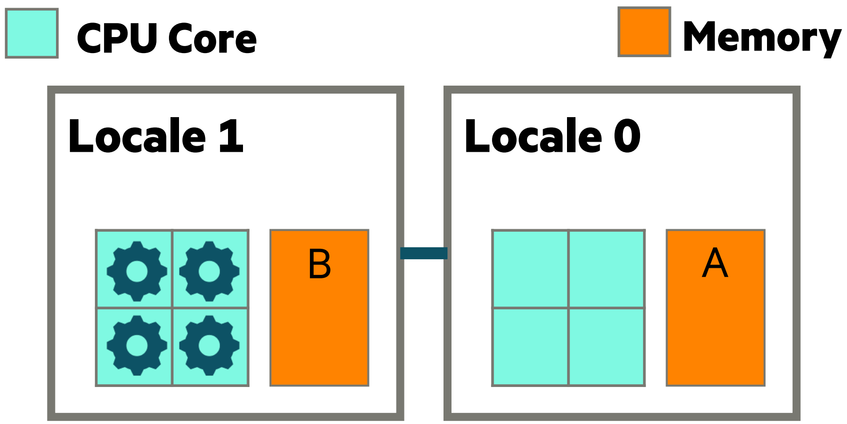
- Execution starts with a single task running on the first locale (i.e. **Locale[0]**)
- Use **coforall** loops to spawn new tasks (one per loop iteration); tasks synchronize at end of loop
- Use **on** statements to move an executing task from one locale to another
- The Locales array contains locales for all the nodes in your system




```
// Execution starts on Locale[0]
var A: [1..4] real;
coforall i in 1..4 do
  A[i] = someComputation(i);
on Locales[1] {
  var B: [1..4] real;
  B = 2;
  A = B;
}
```

Using Locales in Chapel

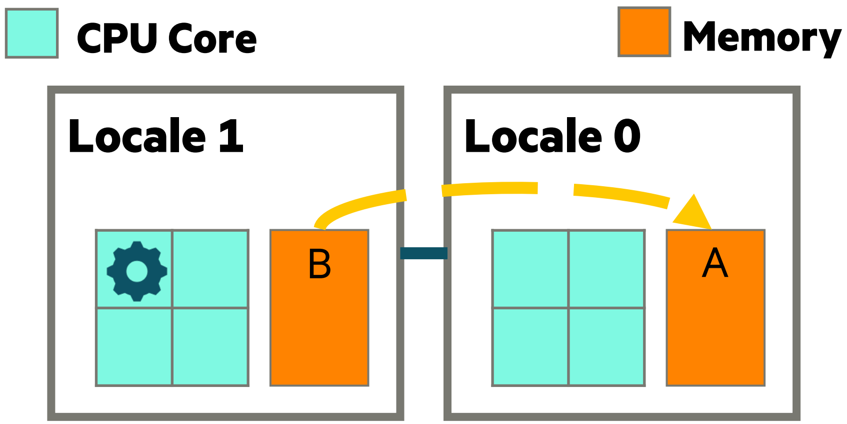
- Execution starts with a single task running on the first locale (i.e. **Locale[0]**)
- Use **coforall** loops to spawn new tasks (one per loop iteration); tasks synchronize at end of loop
- Use **on** statements to move an executing task from one locale to another
- The Locales array contains locales for all the nodes in your system




```
// Execution starts on Locale[0]
var A: [1..4] real;
coforall i in 1..4 do
  A[i] = someComputation(i);
on Locales[1] {
  var B: [1..4] real;
  B = 2;
  A = B;
}
```

Using Locales in Chapel

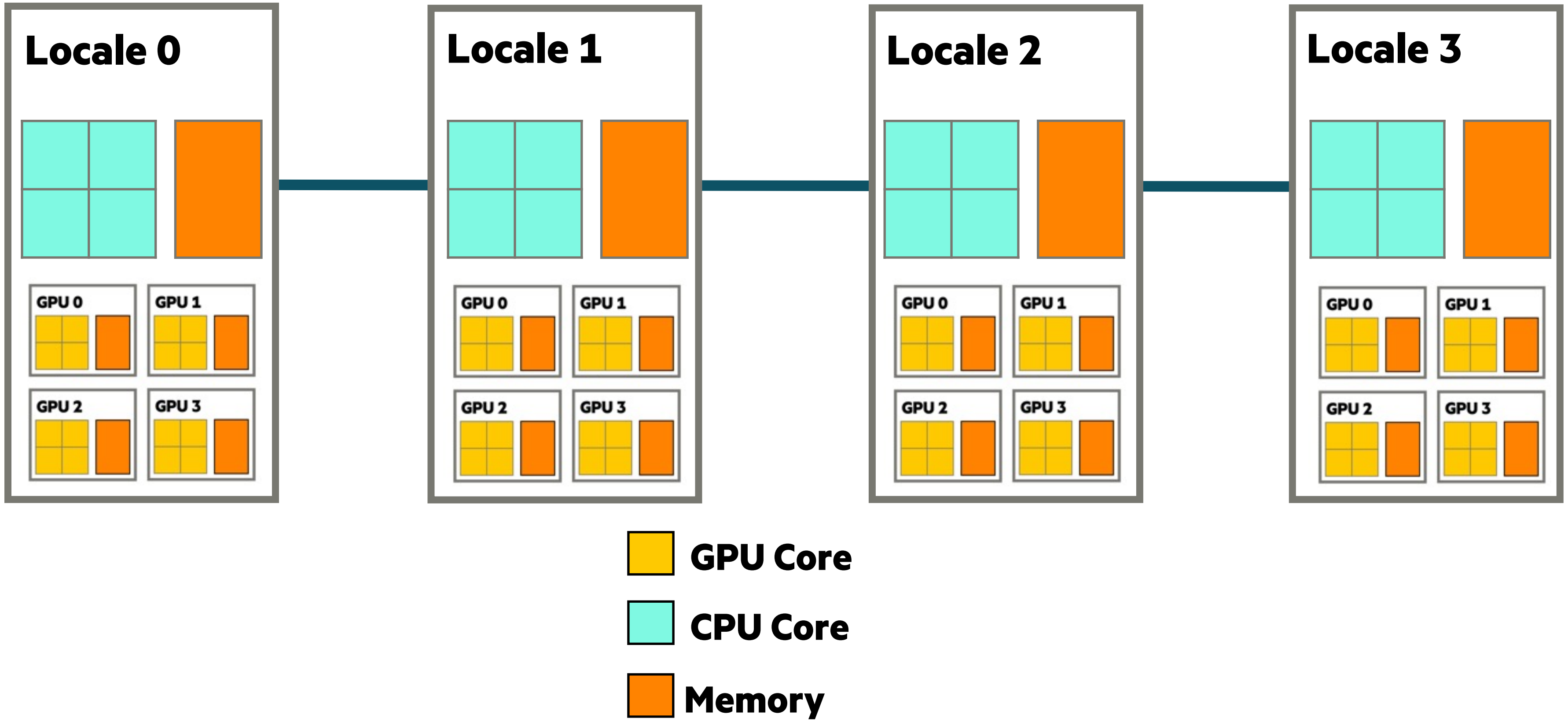
- Execution starts with a single task running on the first locale (i.e. **Locale[0]**)
- Use **coforall** loops to spawn new tasks (one per loop iteration); tasks synchronize at end of loop
- Use **on** statements to move an executing task from one locale to another
- The Locales array contains locales for all the nodes in your system



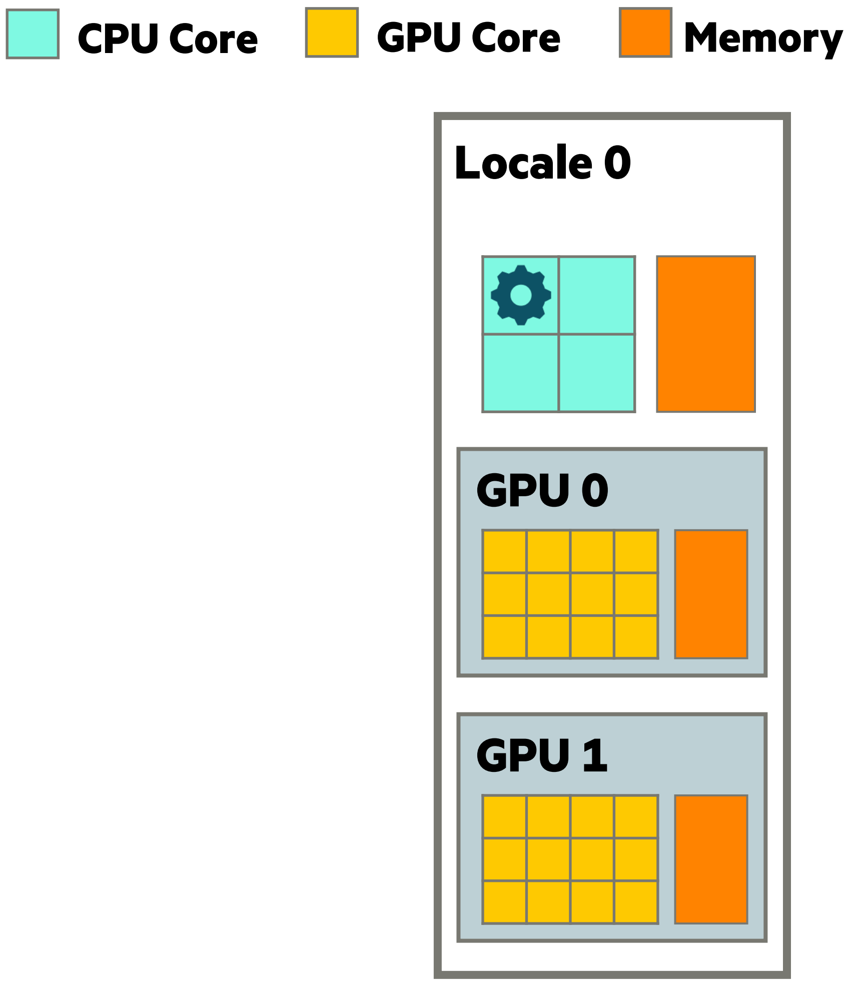

```
// Execution starts on Locale[0]
var A: [1..4] real;
coforall i in 1..4 do
  A[i] = someComputation(i);
on Locales[1] {
  var B: [1..4] real;
  B = 2;
  A = B;
}
```

GPU Sublocales

- Let's add GPU sublocales to the picture
  - These are nested under top level node locales
- Refer to gpu sublocales using the **gpus** array accessible from top-level locales



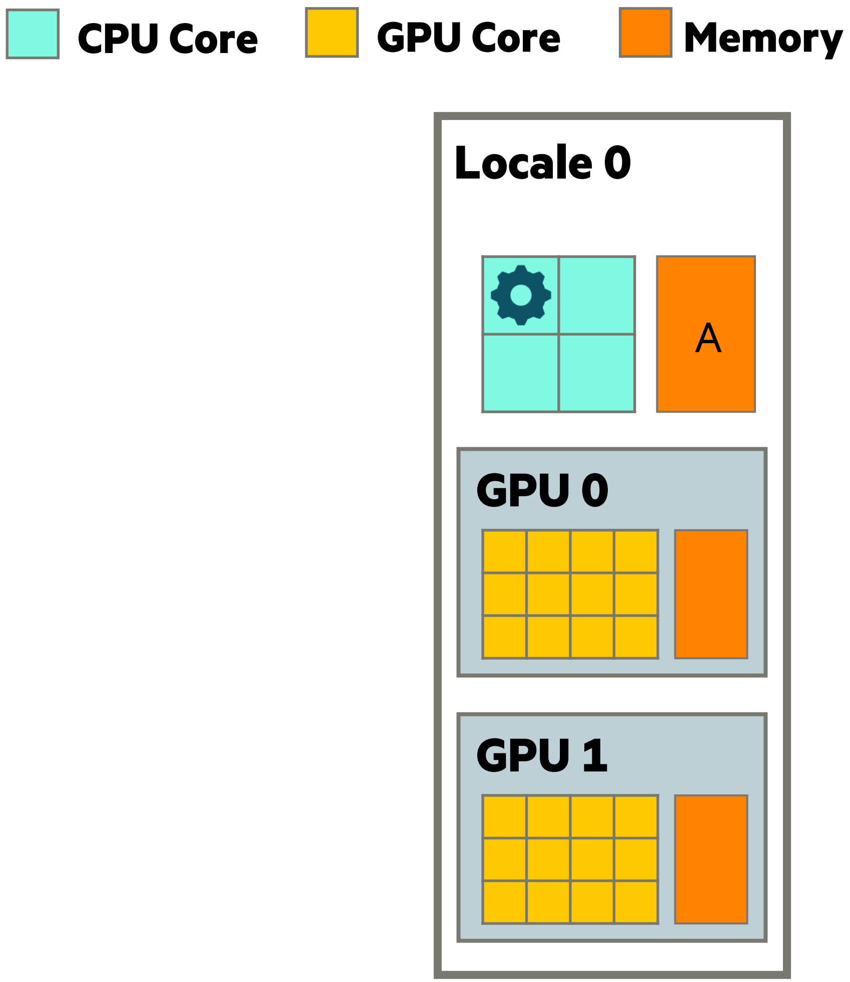
Parallelism and Locality In The Context Of GPUs




```
// Execution starts on Locale[0]
var A: [1..4] real;


  on here.gpus[0] {
   var B: [1..4] real;
   B = A;
   foreach i in 1..4 do
     b[i] = someComputation(i);
   A = B;
  }
```

Parallelism and Locality In The Context Of GPUs

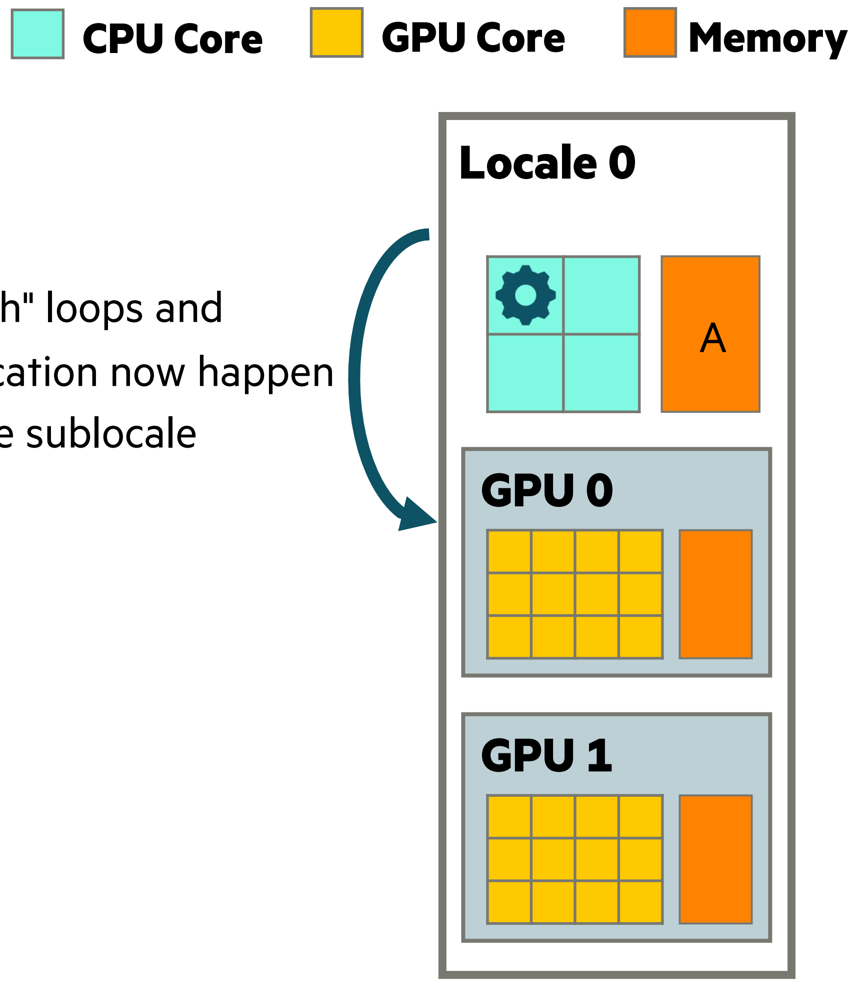



```
// Execution starts on Locale[0]
var A: [1..4] real;


  on here.gpus[0] {
   var B: [1..4] real;
   B = A;
   foreach i in 1..4 do
     b[i] = someComputation(i);
   A = B;
  }
```

Parallelism and Locality In The Context Of GPUs

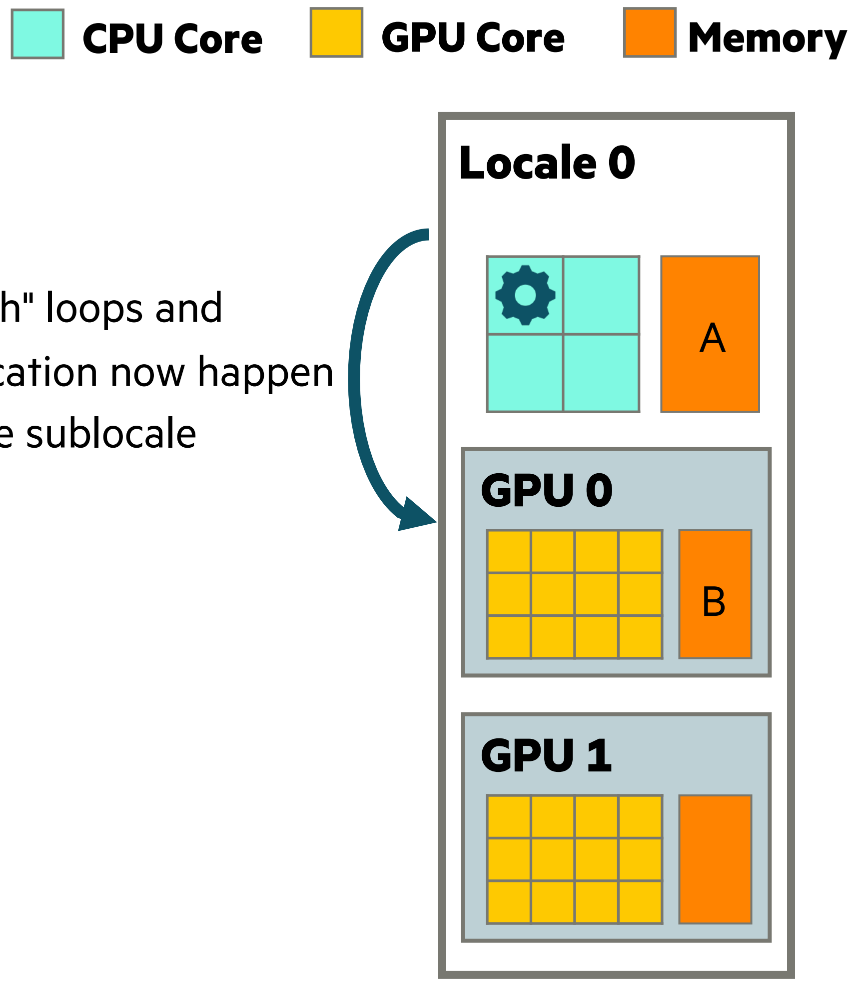
"foreach" loops and memory allocation now happen on the sublocale




```
// Execution starts on Locale[0]
var A: [1..4] real;

on here.gpus[0] {
  var B: [1..4] real;
  B = A;
  foreach i in 1..4 do
    b[i] = someComputation(i);
  A = B;
}
```

Parallelism and Locality In The Context Of GPUs



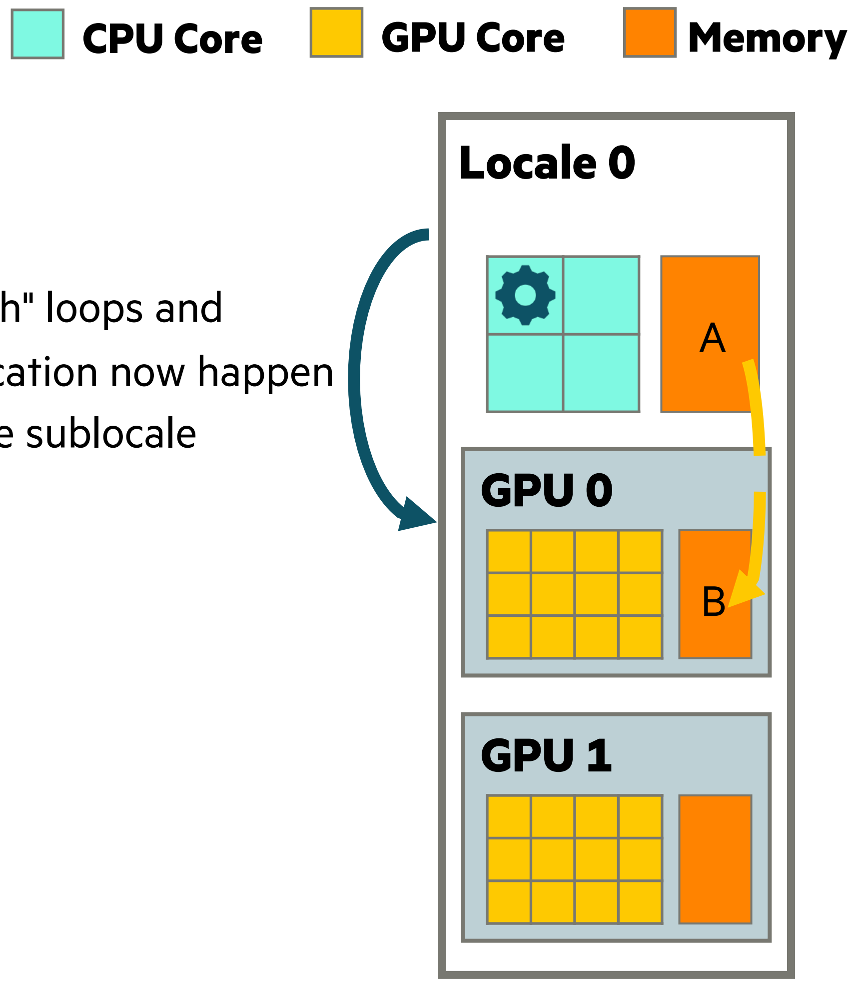
"foreach" loops and
memory allocation now happen
on the sublocale

```
// Execution starts on Locale[0]
var A: [1..4] real;


  on here.gpus[0] {
   var B: [1..4] real;
   B = A;
   foreach i in 1..4 do
     b[i] = someComputation(i);
   A = B;
  }
```

Parallelism and Locality In The Context Of GPUs

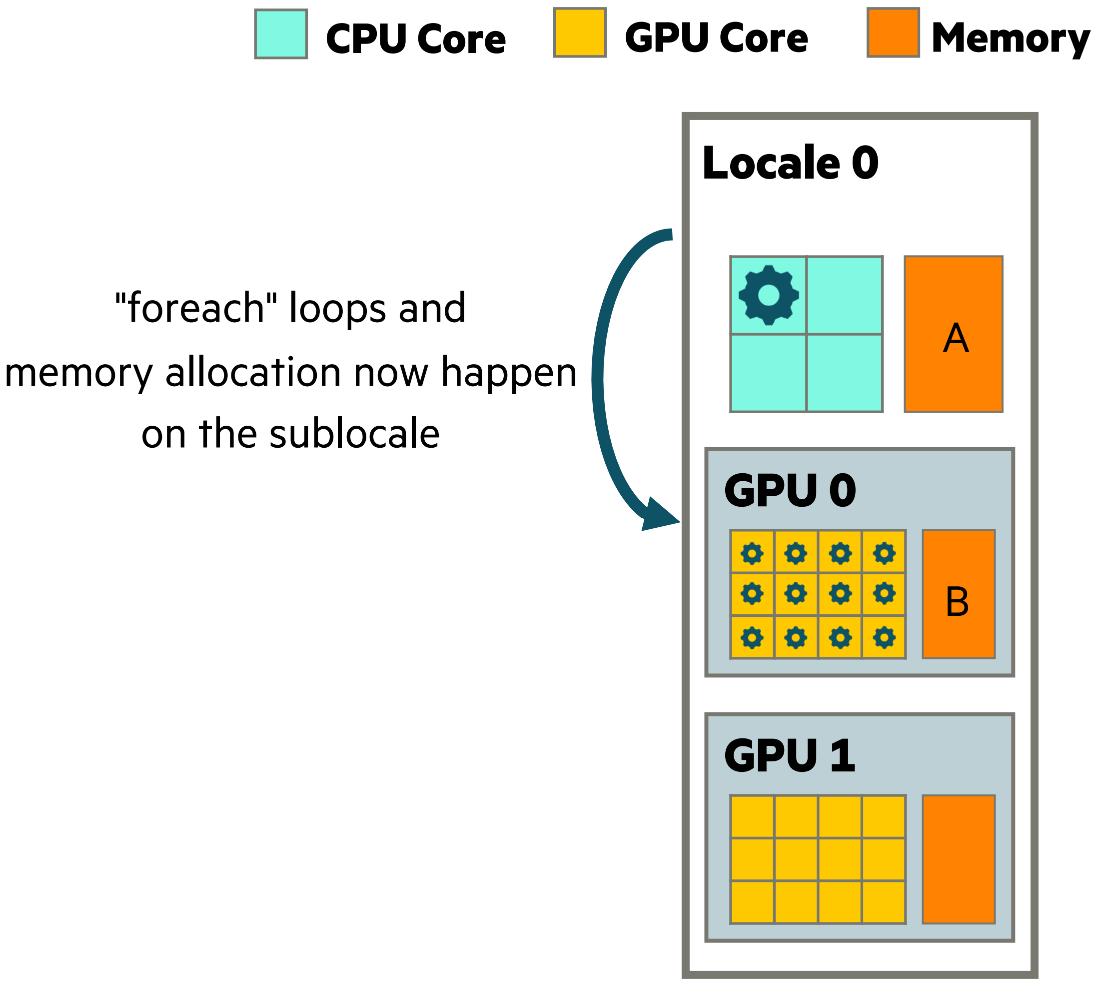
"foreach" loops and memory allocation now happen on the sublocale




```
// Execution starts on Locale[0]
var A: [1..4] real;


  on here.gpus[0] {
   var B: [1..4] real;
   B = A;
   foreach i in 1..4 do
     b[i] = someComputation(i);
   A = B;
  }
```

Parallelism and Locality In The Context Of GPUs

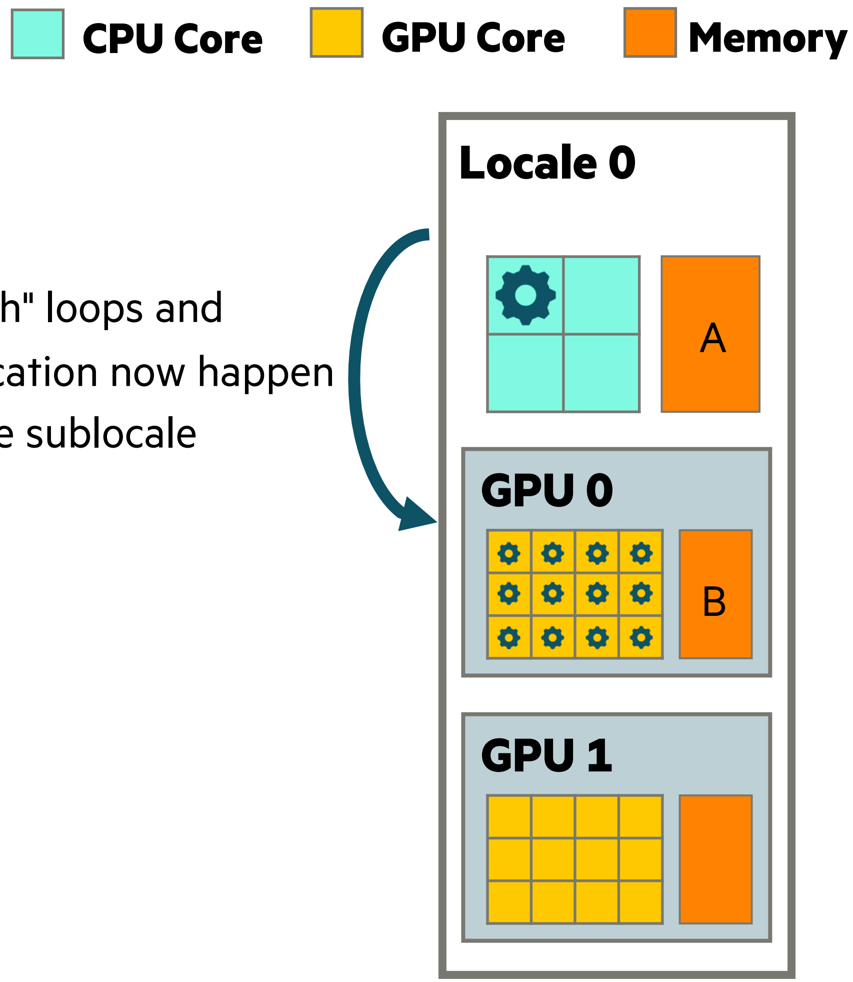



```
// Execution starts on Locale[0]
var A: [1..4] real;


  on here.gpus[0] {
   var B: [1..4] real;
   B = A;
   foreach i in 1..4 do
     b[i] = someComputation(i);
   A = B;
  }
```

Parallelism and Locality In The Context Of GPUs

"foreach" loops and memory allocation now happen on the sublocale

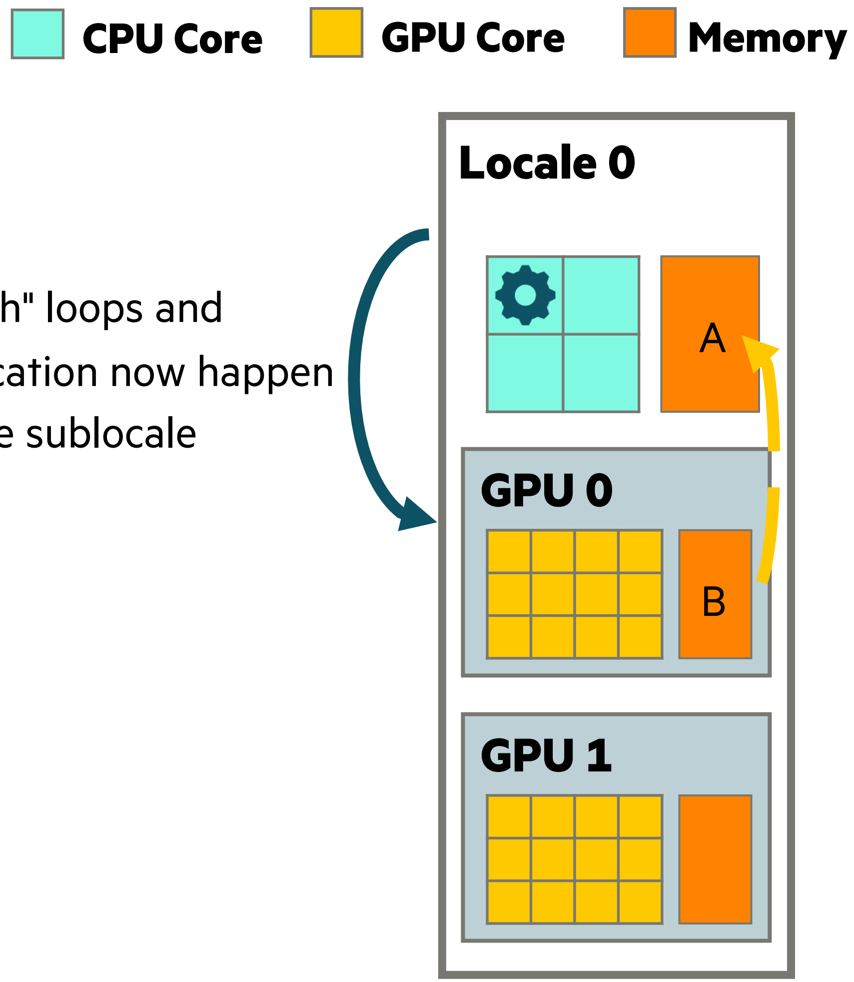



```
// Execution starts on Locale[0]
var A: [1..4] real;


  on here.gpus[0] {
   var B: [1..4] real;
   B = A;
   foreach i in 1..4 do
     b[i] = someComputation(i);
   A = B;
  }
```

Parallelism and Locality In The Context Of GPUs

"foreach" loops and memory allocation now happen on the sublocale

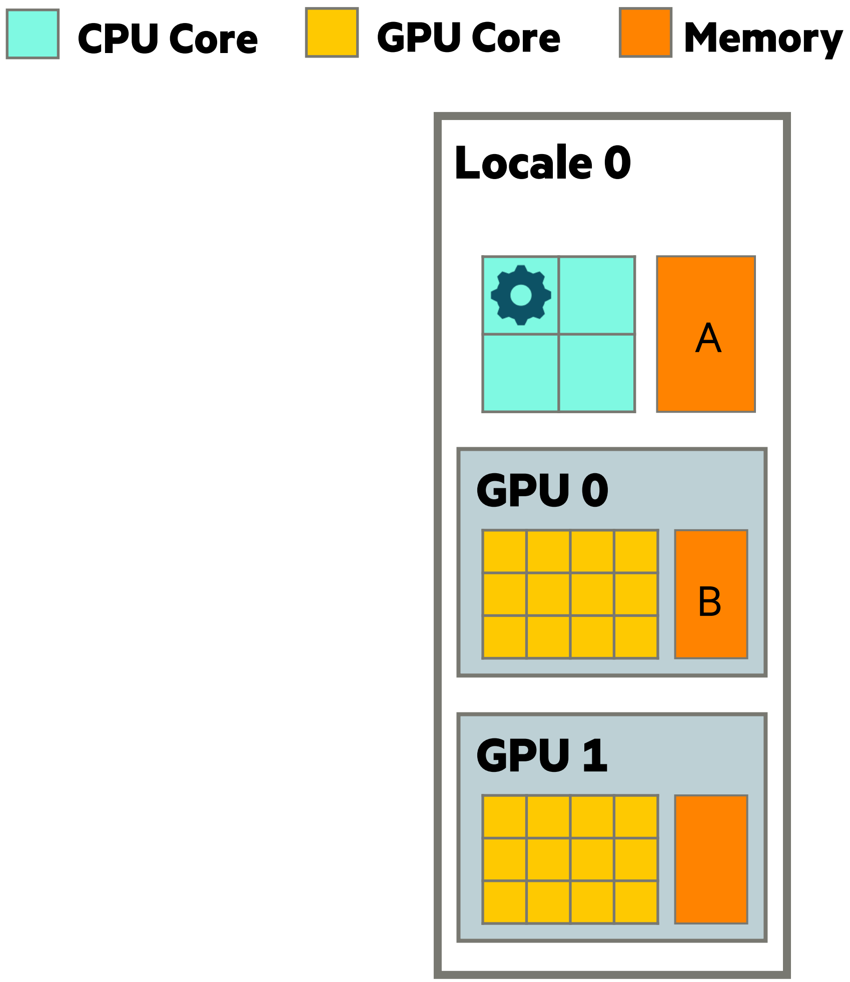


// Execution starts on Locale[0]

```
var A: [1..4] real;


  on here.gpus[0] {
   var B: [1..4] real;
   B = A;
   foreach i in 1..4 do
     b[i] = someComputation(i);
   A = B;
  }
```

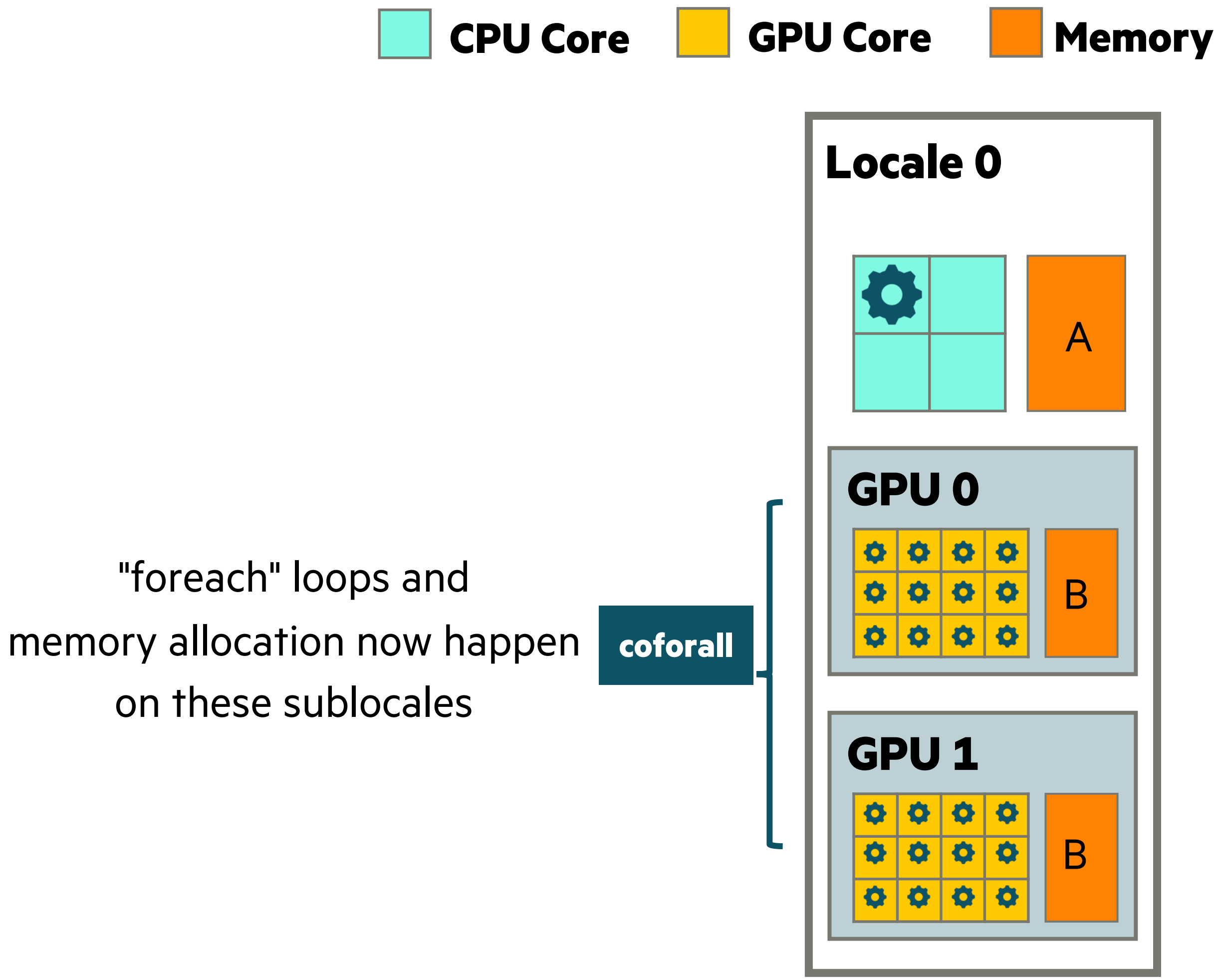
Parallelism and Locality In The Context Of GPUs




```
// Execution starts on Locale[0]
var A: [1..4] real;


  on here.gpus[0] {
   var B: [1..4] real;
   B = A;
   foreach i in 1..4 do
     b[i] = someComputation(i);
   A = B;
  }
```

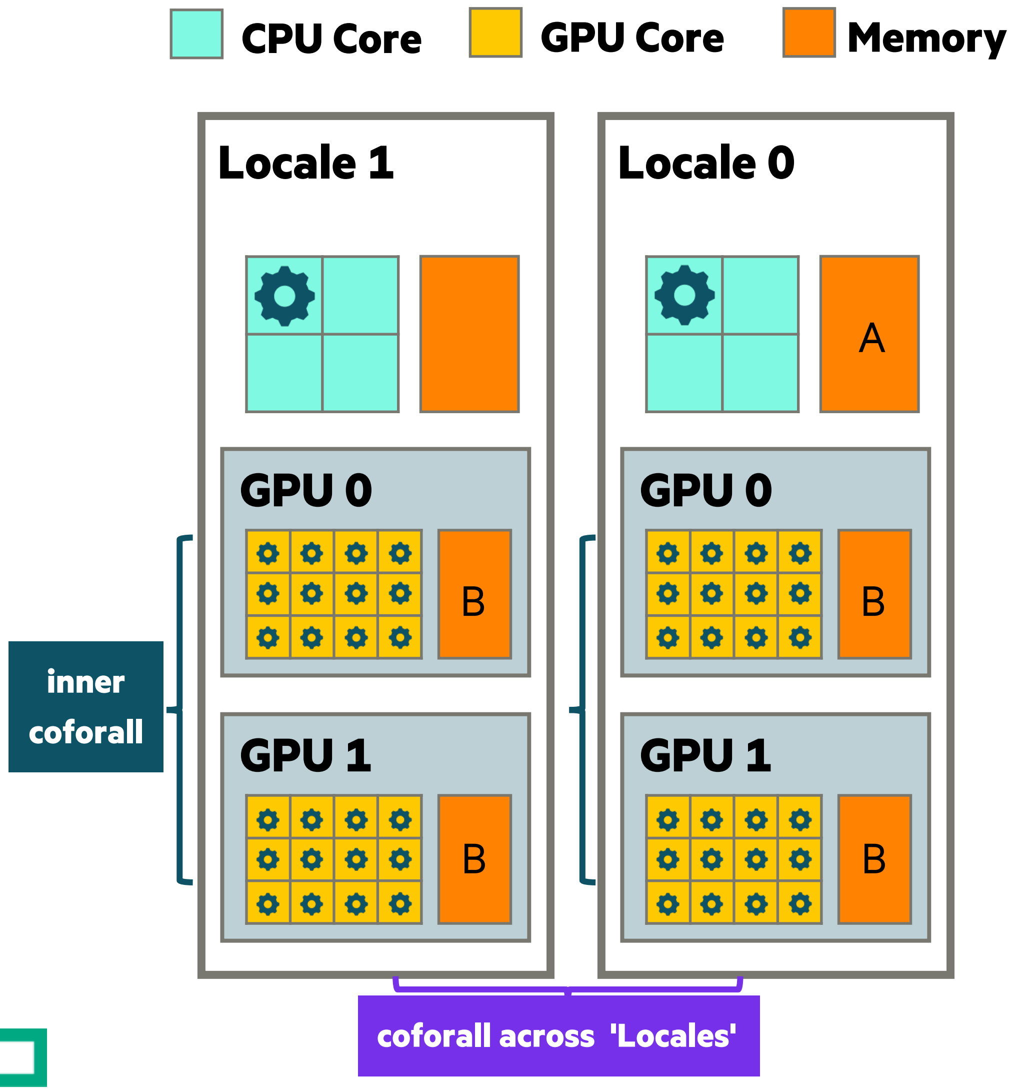
Parallelism and Locality In The Context Of GPUs




```
// Execution starts on Locale[0]
var A: [1..4] real;

coforall g in here.gpus do on g {
  var B: [1..4] real;
  B = A;
  foreach i in 1..4 do
    b[i] = someComputation(i);
  A = B;
}
```

Parallelism and Locality In The Context Of GPUs




```
// Execution starts on Locale[0]
var A: [1..4] real;
coforall loc in Locales do on loc {

  coforall g in here.gpus do on g {
    var B: [1..4] real;
    B = A;
    foreach i in 1..4 do
      b[i] = someComputation(i);
    A = B;
  }
}
```

Conclusions

Conclusions

- Chapel enables scalable programming programming
- Chapel represents nodes as locales and GPUs as sublocales
- We presented a handful of community-written applications
  - These work both on NVIDIA and AMD GPUs without any vendor-specific code modifications
  - Chapel performs competitively for most, we're working on Tealeaf
- GPU support in Chapel has garnered a lot of interest
- Future work includes support for Chapel's distributed arrays on GPUs and execution on Intel GPUs

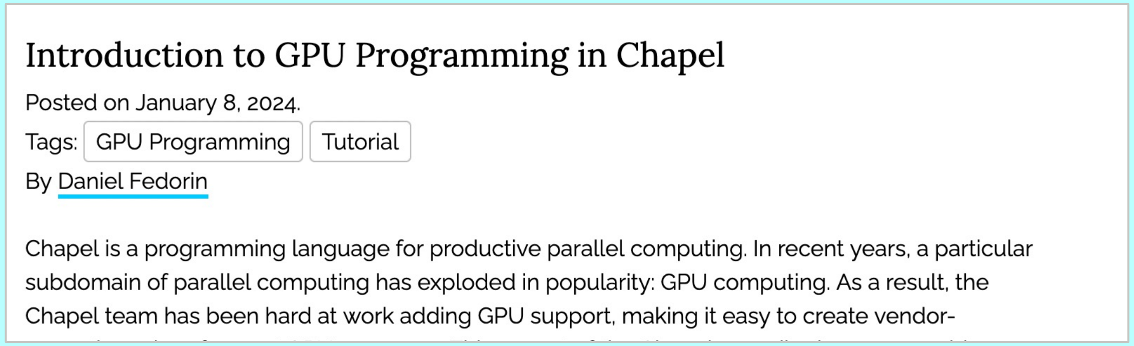
If you Want to Learn More About GPU Programming In Chapel

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

Blogpost: https://chapel-lang.org/blog/posts/intro-to-gpus/

- Tutorial on GPU programming in Chapel


Introduction to GPU Programming in Chapel

Posted on January 8, 2024.

Tags: GPU Programming Tutorial

By Daniel Fedorin

Chapel is a programming language for productive parallel computing. In recent years, a particular subdomain of parallel computing has exploded in popularity: GPU computing. As a result, the Chapel team has been hard at work adding GPU support, making it easy to create vendor-

Previous talks

- **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
- **Recent Release Notes:** almost everything that happened in each release
  - https://chapel-lang.org/release-notes-archives.html

Chapel Resources

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

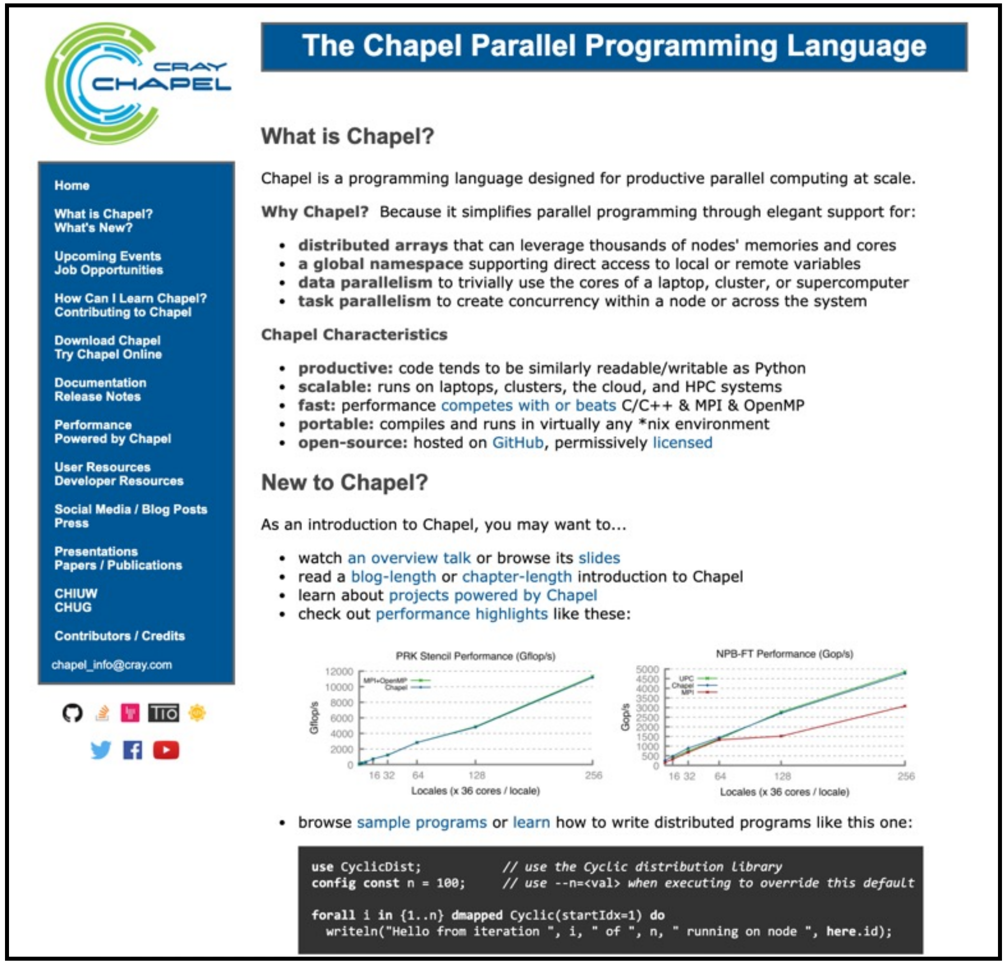
- (points to all other resources)

Social Media:

- Twitter: @ChapelLanguage
- Facebook: @ChapelLanguage
- YouTube: https://www.youtube.com/c/ChapelParallelProgrammingLanguage
- Blog: https://chapel-lang.org/blog/

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



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

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

The Chapel Event of the Year!

June 5–7, 2024
Free and online in a virtual format

Tutorials, open lab sessions, demos, and talks

Conclusions

- Chapel enables scalable, parallel, programming
- Chapel represents nodes as locales and GPUs as sublocales
- We presented a handful of community-written applications
  - These work both on NVIDIA and AMD GPUs without any vendor-specific code modifications
  - Chapel performs competitively for most, we're working on Tealeaf
- GPU support in Chapel has garnered a lot of interest
- Future work includes support for Chapel's distributed arrays on GPUs and execution on Intel GPUs

ChapelCon
June 5–7, 2024
Free and online in a virtual format
https://chapel-lang.org/ChapelCon24.html
Tutorials, open lab sessions, demos, and talks