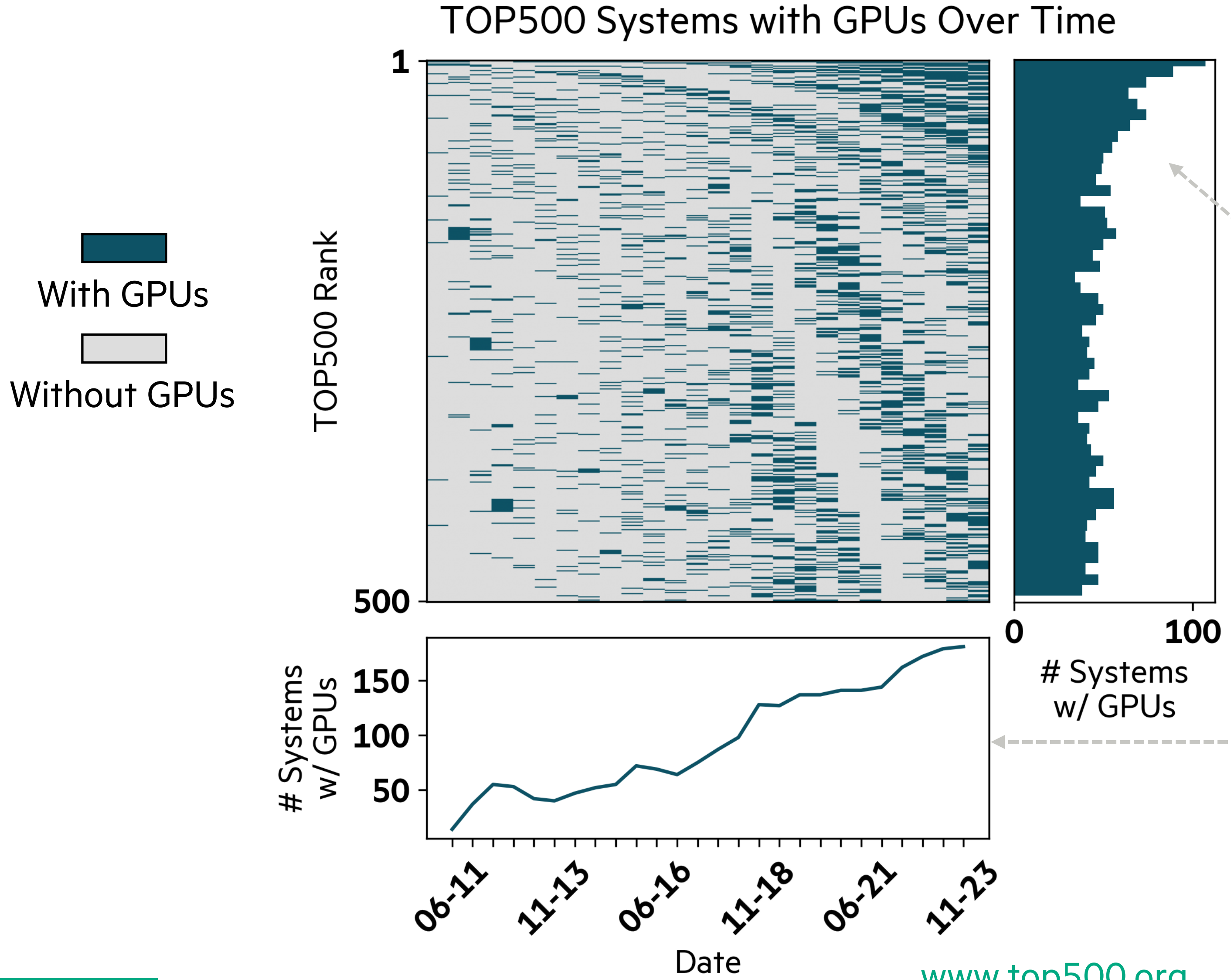
Hewlett Packard
Enterprise

HIGH-LEVEL, VENDOR-NEUTRAL GPU PROGRAMMING USING CHAPEL

Engin Kayraklioglu
January 9th, 2024

IT IS HARD TO AVOID GPUS IN HPC



www.top500.org

41% of **top 10** have GPUs

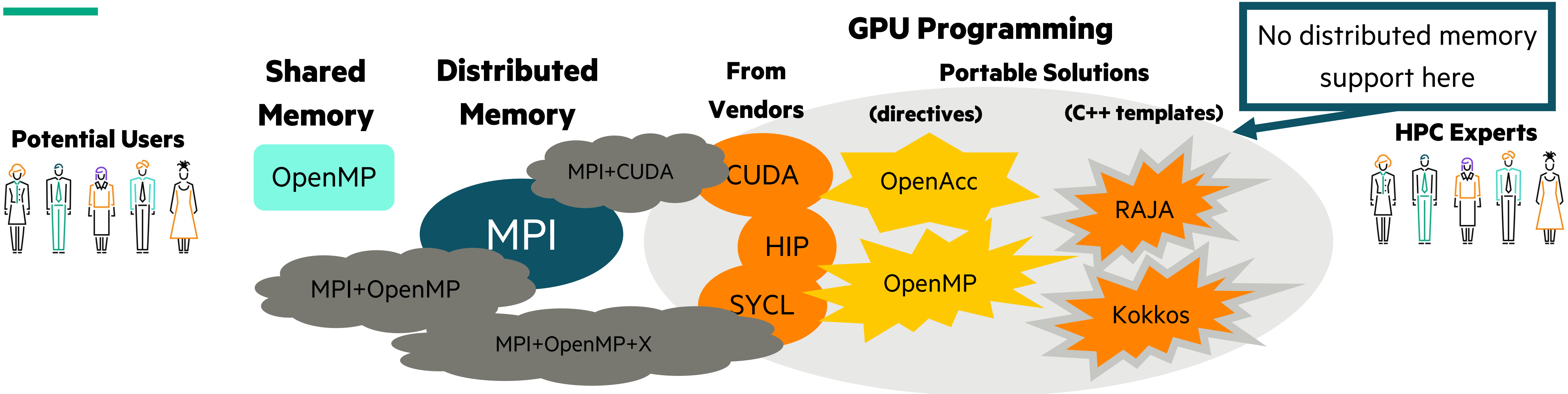
72% in the last **3 years**

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



Potential Users

Shared Memory

OpenMP

Distributed Memory

MPI

MPI+CUDA

MPI+OpenMP

MPI+OpenMP+X

GPU Programming

From Vendors

CUDA

HIP

SYCL

Portable Solutions

(directives)

OpenAcc

OpenMP

(C++ templates)

RAJA

Kokkos

No distributed memory support here

HPC Experts

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
- runs on Linux laptops/clusters, Cray systems, MacOS, WSL, AWS, Raspberry Pi
- shown to scale on Cray networks (Slingshot, Aries), InfiniBand, RDMA-Ethernet

Can we expand this list to
GPUs from all vendors?

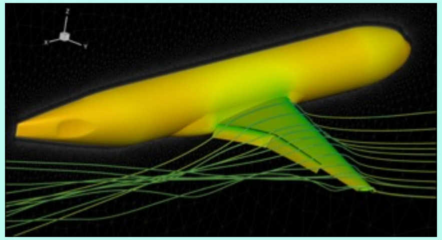
Chapel makes distributed/shared memory parallel programming easy

- data-parallel, locality-aware loops,
- ability to move execution to remote nodes,
- distributed arrays and bulk array operations
- …

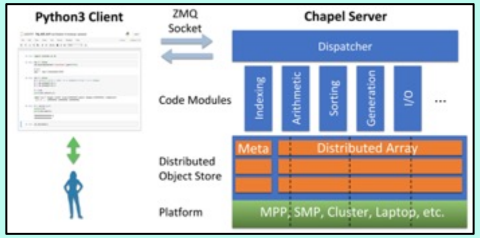
While using the
same expressive features?

APPLICATIONS OF CHAPEL

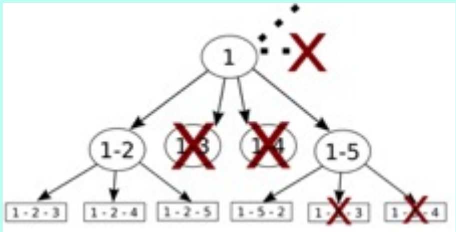
Active GPU efforts



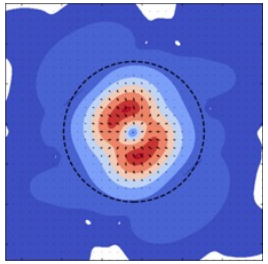
CHAMPS: 3D Unstructured CFD
Laurendeau, Bourgault-Côté, Parenteau, Plante, et al.
École Polytechnique Montréal



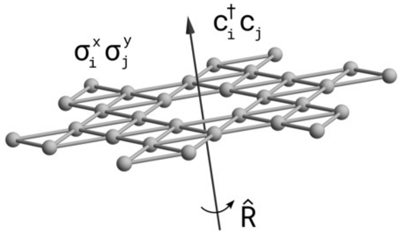
Arkouda: Interactive Data Science at Massive Scale
Mike Merrill, Bill Reus, et al.
U.S. DoD



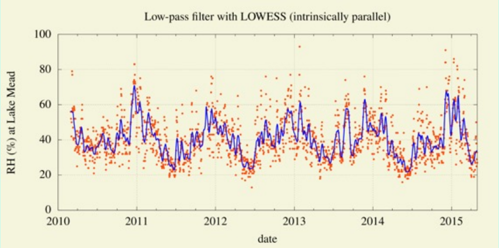
ChOp: Chapel-based Optimization
T. Carneiro, G. Helbecque, N. Melab, et al.
INRIA, IMEC, et al.



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

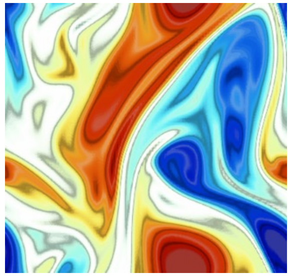


Lattice-Symmetries: a Quantum Many-Body Toolbox
Tom Westerhout
Radboud University

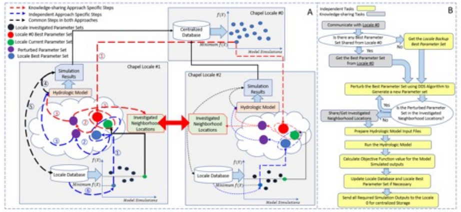


Desk dot chpl: Utilities for Environmental Eng.
Nelson Luis Dias
The Federal University of Paraná, Brazil

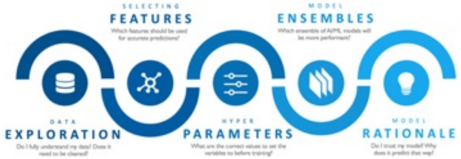
RapidQ: Mapping Coral Biodiversity
Rebecca Green, Helen Fox, Scott Bachman, et al.
The Coral Reef Alliance



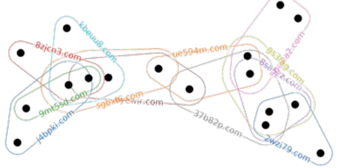
ChapQG: Layered Quasigeostrophic CFD
Ian Grooms and Scott Bachman
University of Colorado, Boulder et al.



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?

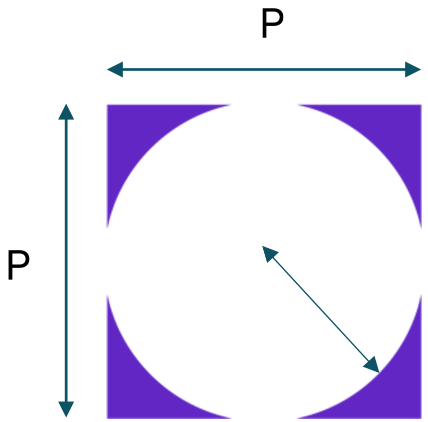
(images provided by their respective teams and used with permission)

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.

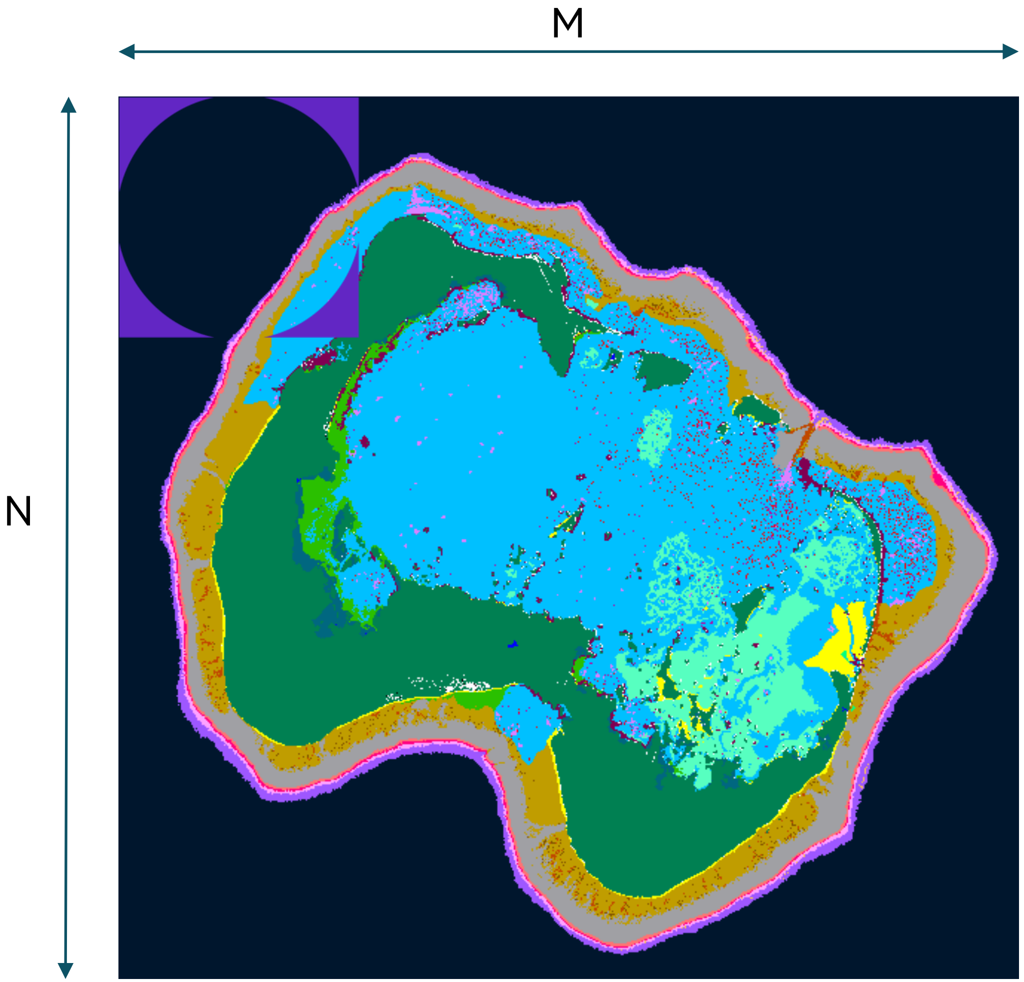


Algorithmic complexity: $O(MNP^3)$

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.

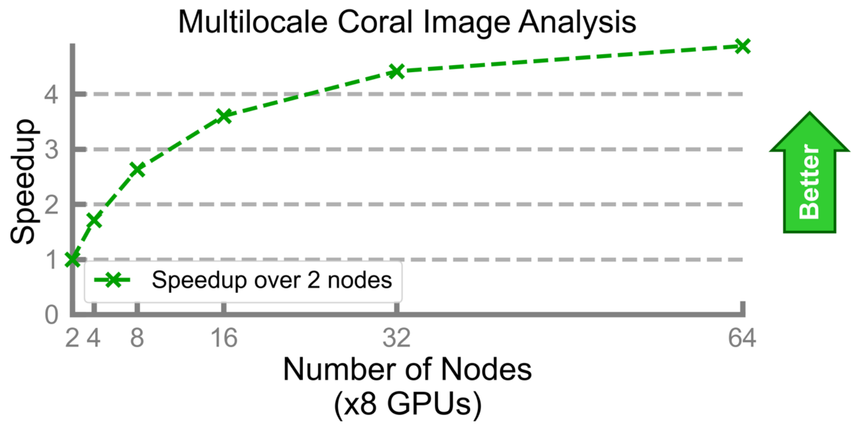
High-level, intuitive array operations work across nodes and/or devices

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;
}}}}
```

Ready to run on multiple nodes 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
- Very high-level overview of how it's implemented in Chapel
- Teasers on ongoing work and future plans

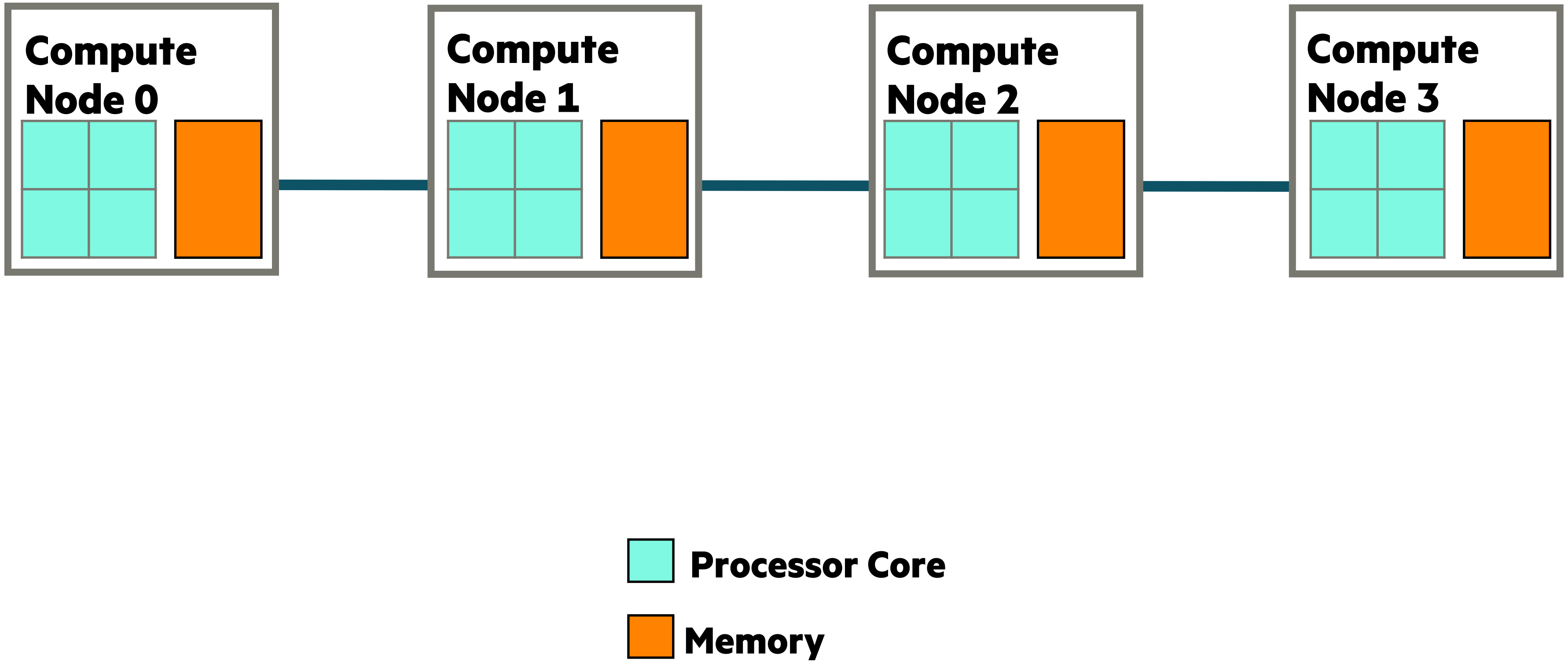
What we will not discuss today:

- Comprehensive list of Chapel features
  - (important ones will be covered)
- Everything you can do with GPUs using Chapel
  - (there's only so much time ☺)

GPU PROGRAMMING IN CHAPEL

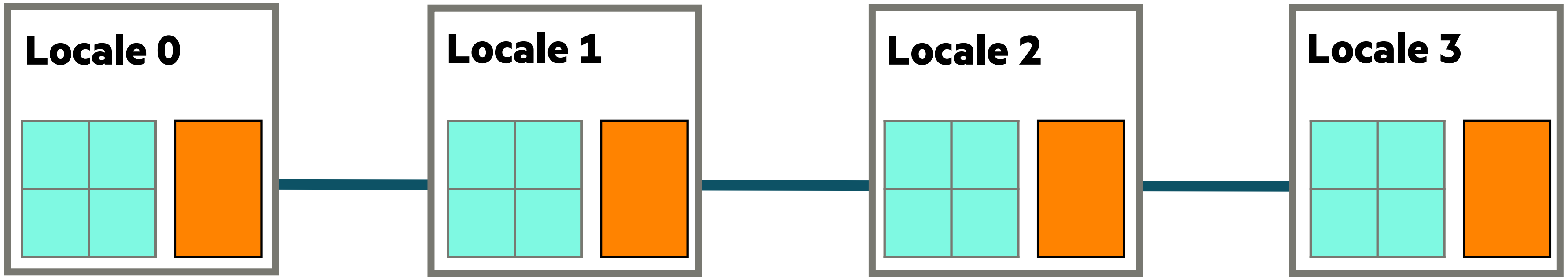
LOCALES IN CHAPEL

- 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



LOCALES IN CHAPEL

- 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

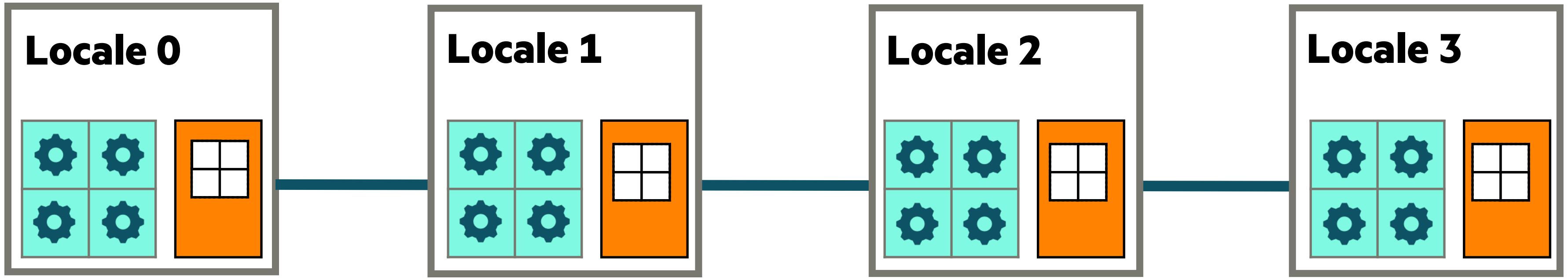


🟦 **Processor Core**

🟧 **Memory**

KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

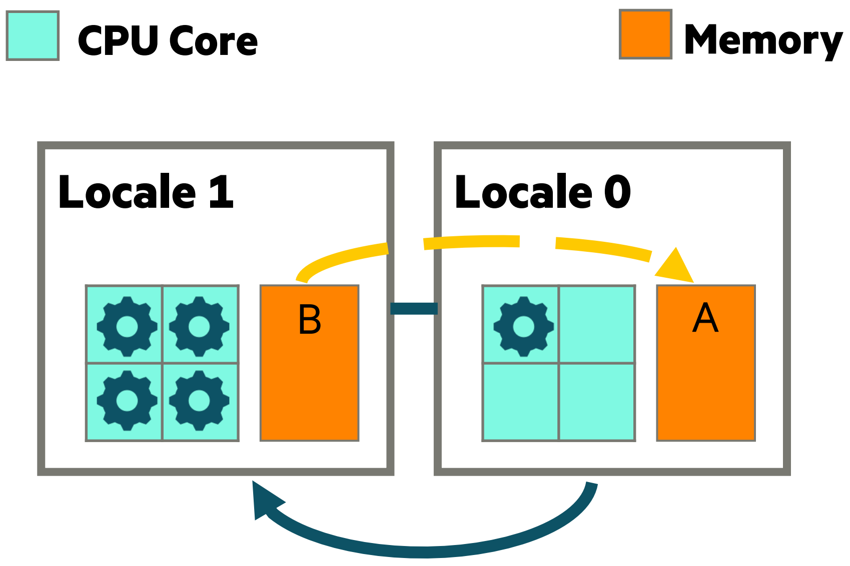
1. **parallelism:** Which tasks should run simultaneously?
2. **locality:** Where should tasks run? Where should data be allocated?



🟦 **Processor Core**

🟧 **Memory**

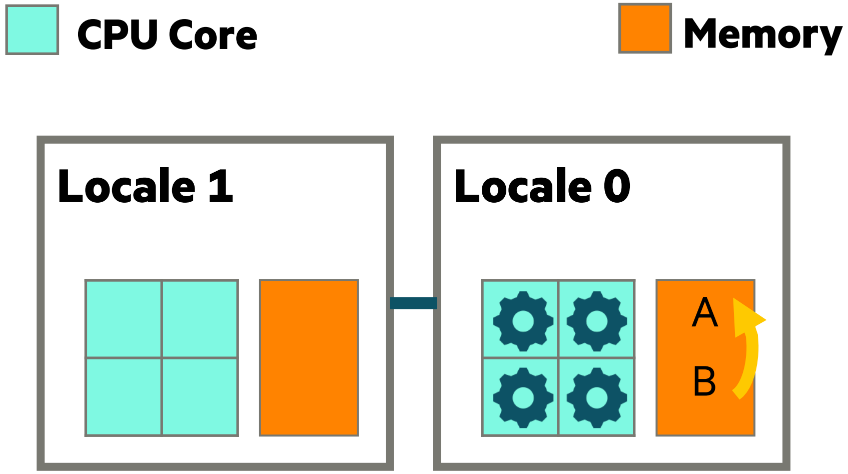
PARALLELISM AND LOCALITY



Execution/allocation moves to Locale 1

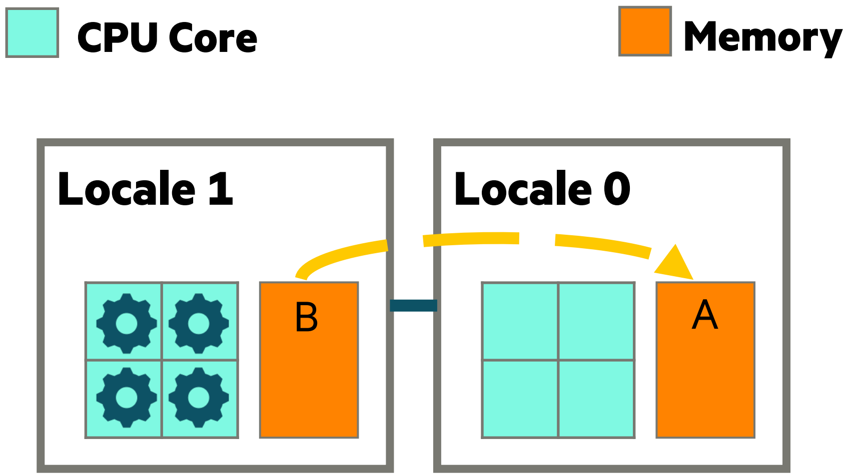
```
var A: [1..2, 1..2] real;


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


writeln(A);
```

PARALLELISM AND LOCALITY



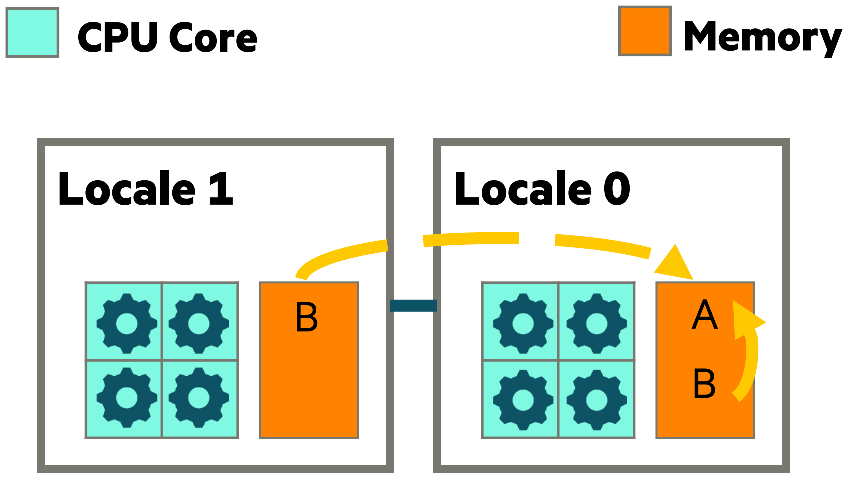
```
var A: [1..2, 1..2] real;


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




writeln(A);
```

PARALLELISM AND LOCALITY



```
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

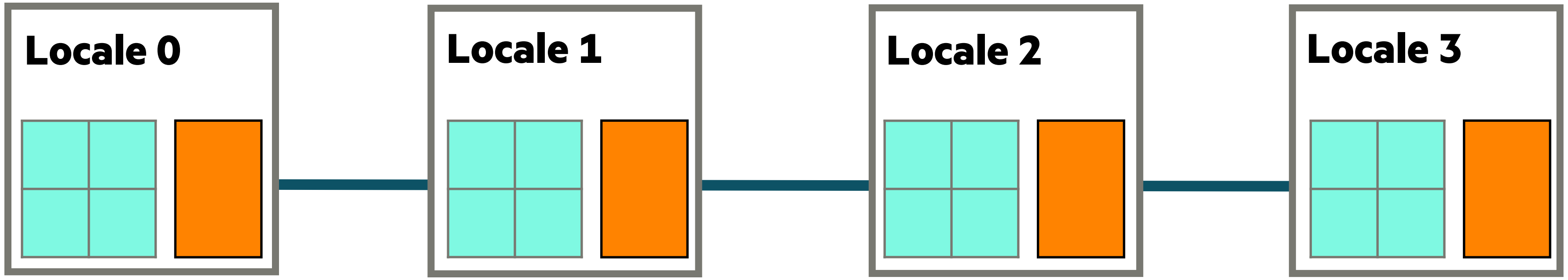


The coforall loop creates
a parallel task per iteration

```
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;
  }




var 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

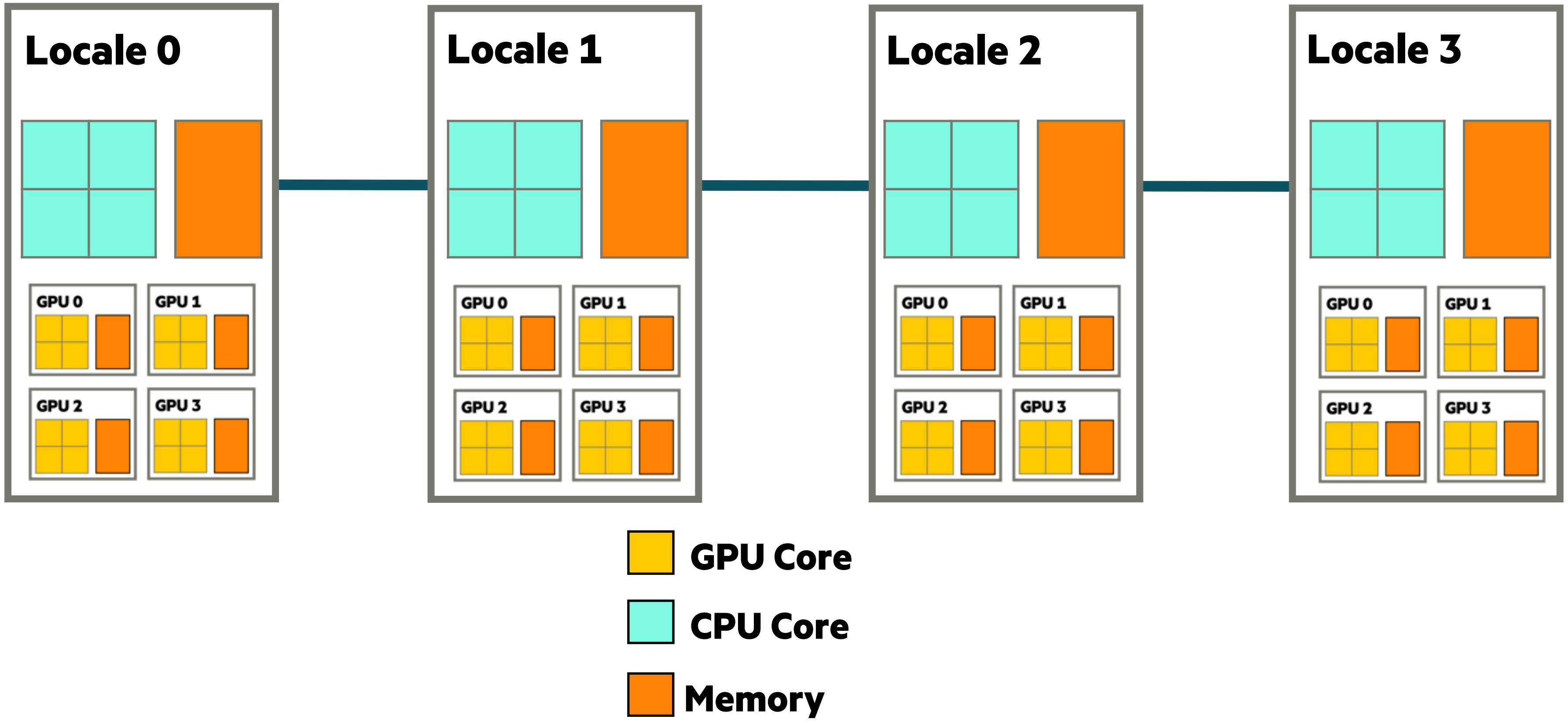


🟩 **Processor Core**

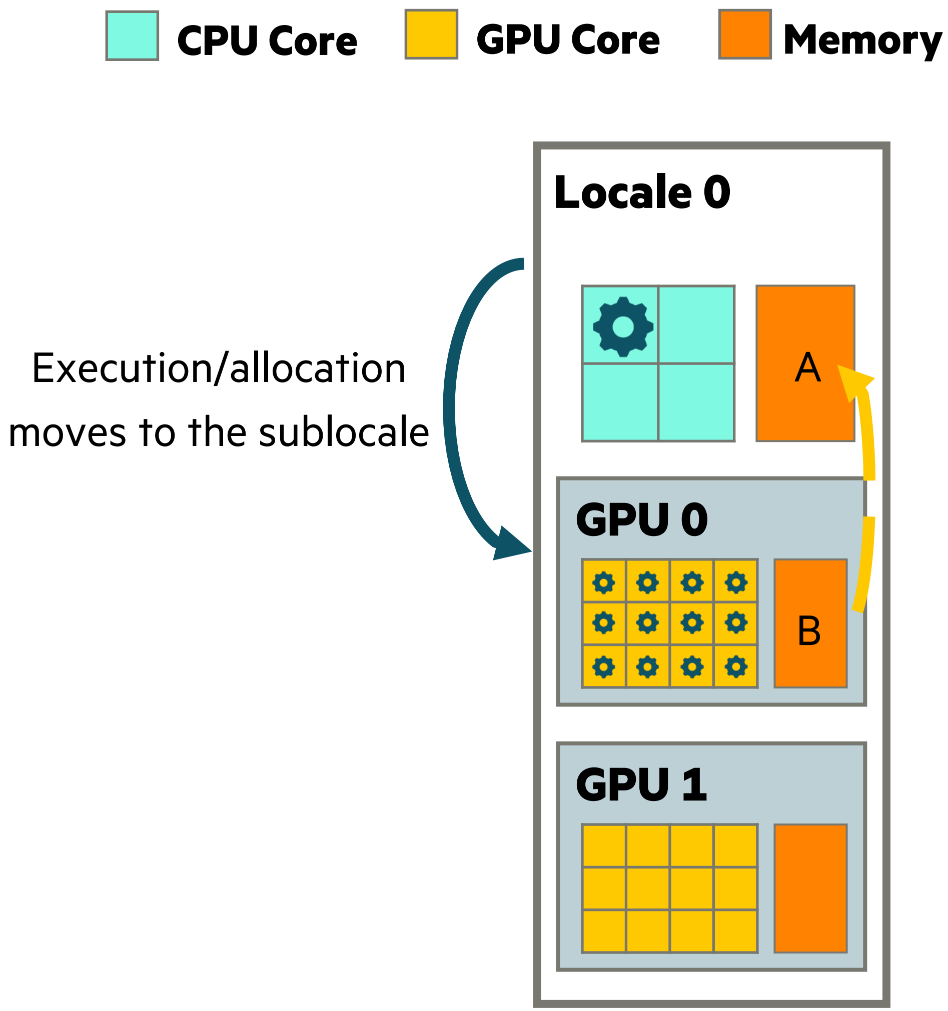
🟧 **Memory**

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



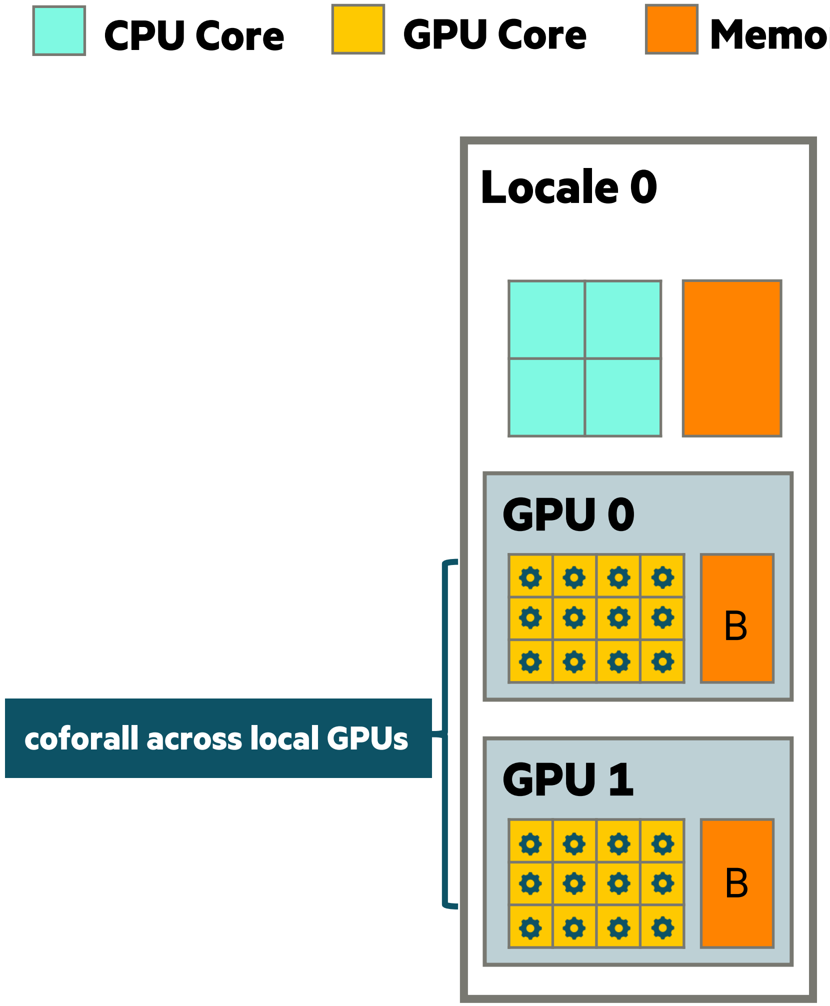
PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS



Execution/allocation moves to the sublocale

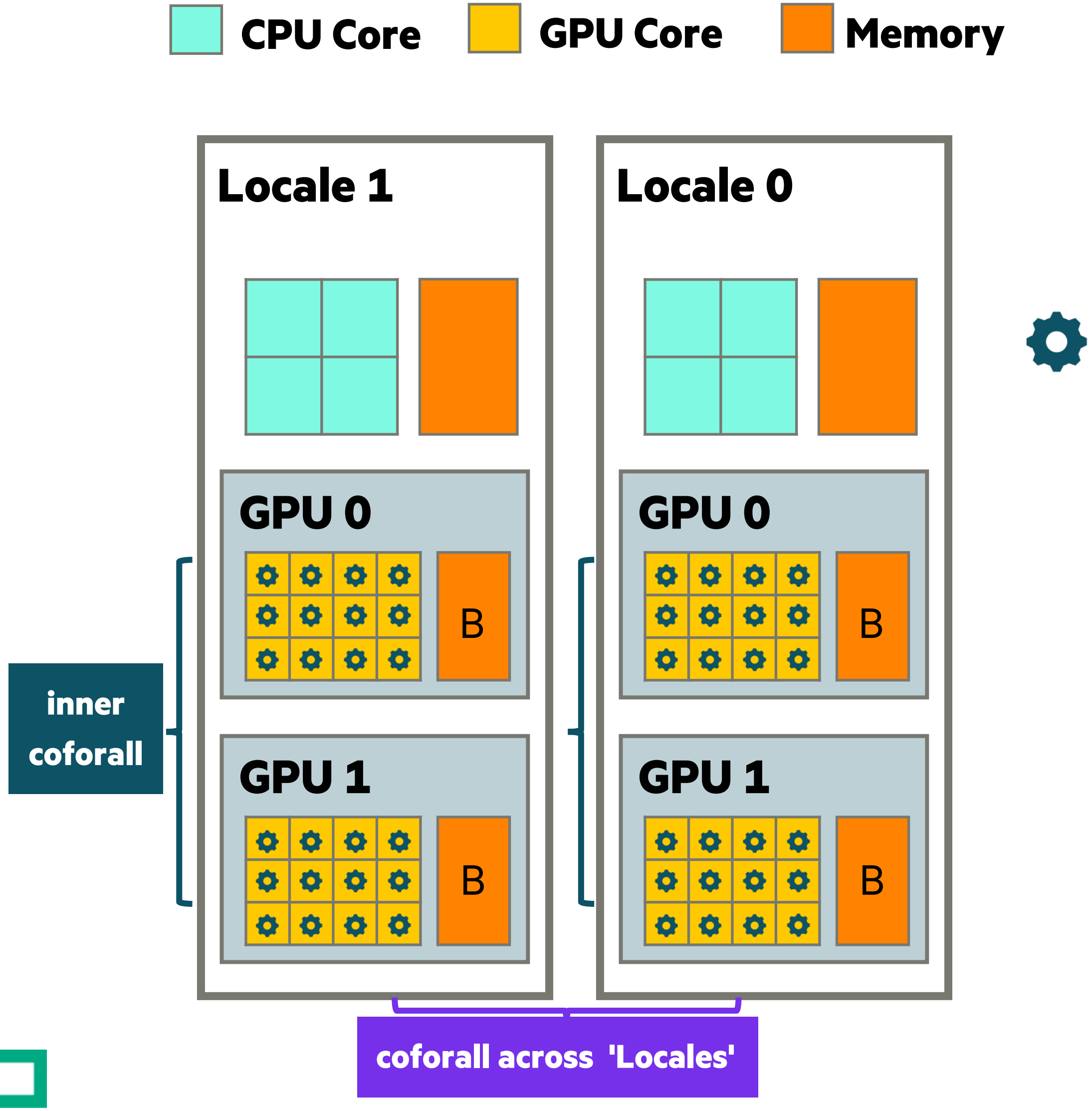
```
var A: [1..2, 1..2] real;


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









writeln(A);
```

PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS



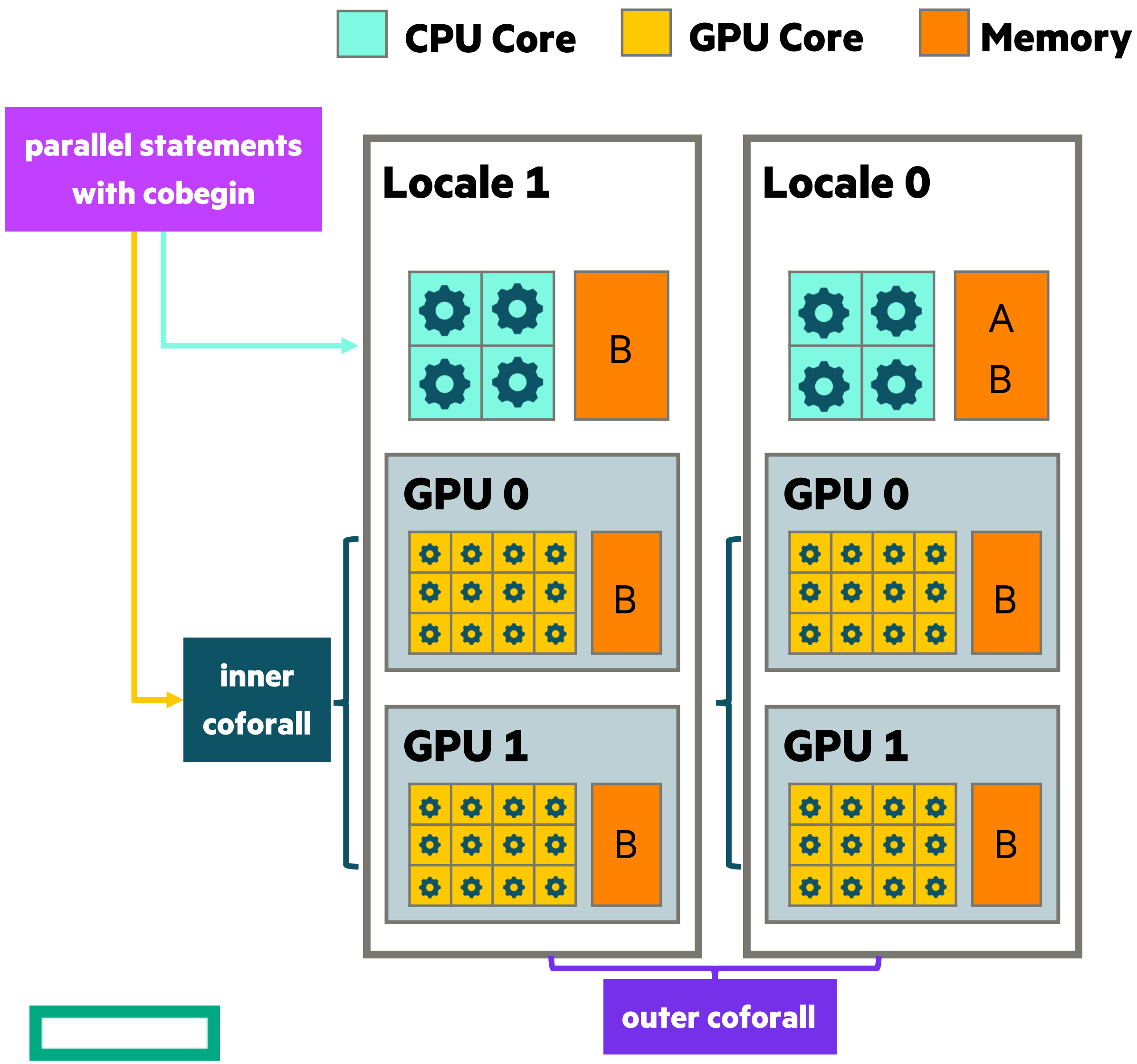
```
var A: [1..2, 1..2] real;

  coforall g in here.gpus do on g {
   var B: [1..2, 1..2] real;
   B = 2;
   A = B;
  }

writeln(A);
```

PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS



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

  coforall g in here.gpus do on g {
   var B: [1..2, 1..2] real;
   B = 2;
   A = B;
  }

}
writeln(A);
```

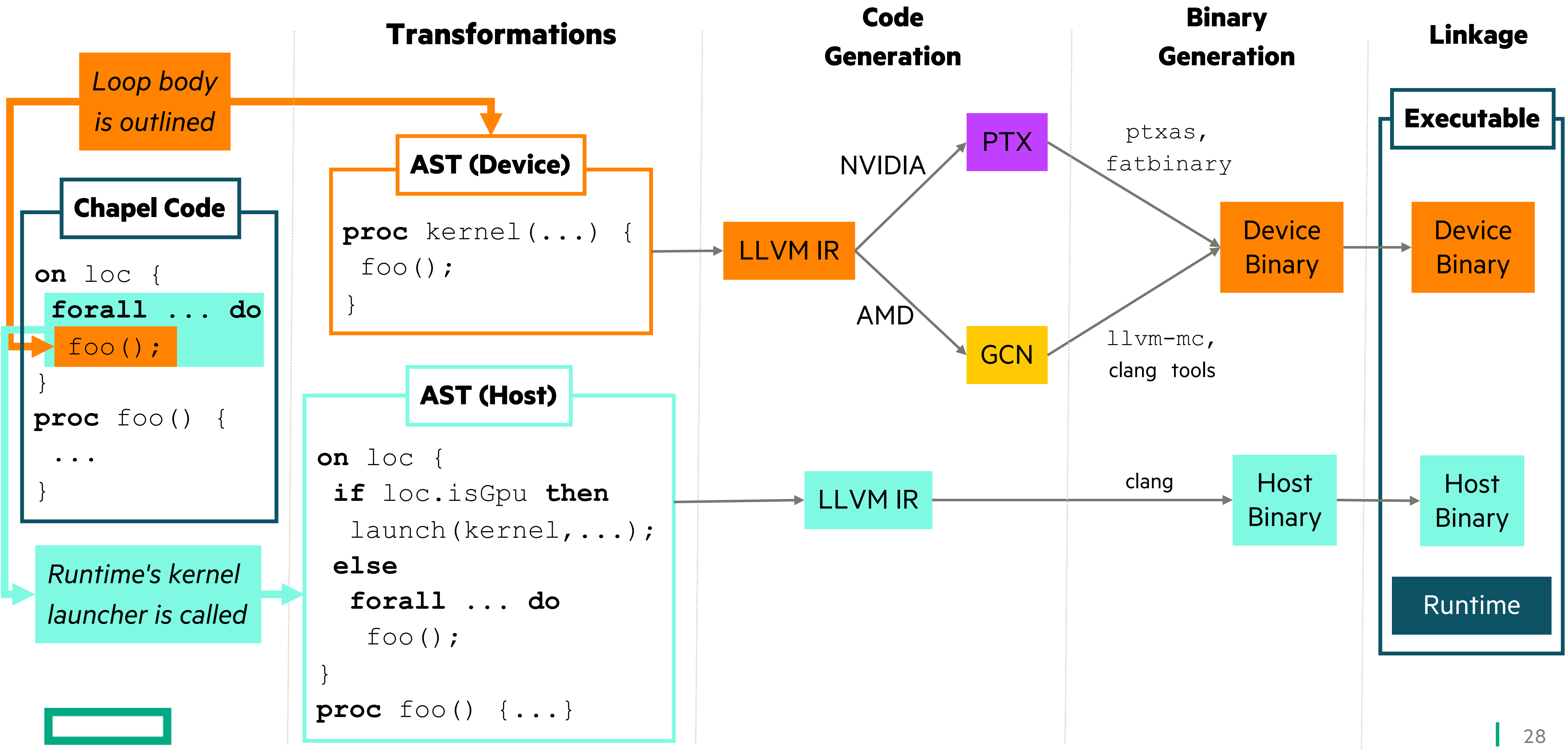
PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS



```
var A: [1..2, 1..2] real;
coforall l in Locales do on l {
 cobegin {
  coforall g in here.gpus do on g {
   var B: [1..2, 1..2] real;
   B = 2;
   A = B;
  }
  {
   var B: [1..2, 1..2] real;
   B = 2;
   A = B;
  }
 }
}
writeln(A);
```

HOW DOES IT WORK?

COMPILATION TRAJECTORY



Transformations

Code Generation

Binary Generation

Linkage

Loop body is outlined

Chapel Code

```
on loc {
 forall ... do
  foo();
}
proc foo() {
 ...
}
```

Runtime's kernel launcher is called

AST (Device)

```
proc kernel(...) {
 foo();
}
```

AST (Host)

```
on loc {
 if loc.isGpu then
  launch(kernel,...);
 else
  forall ... do
   foo();
}
proc foo() {...}
```

LLVM IR

NVIDIA

PTX

AMD

GCN

`ptxas, fatbinary`

`llvm-mc,` clang tools

Device Binary

LLVM IR

clang

Host Binary

Executable

Device Binary

Host Binary

Runtime

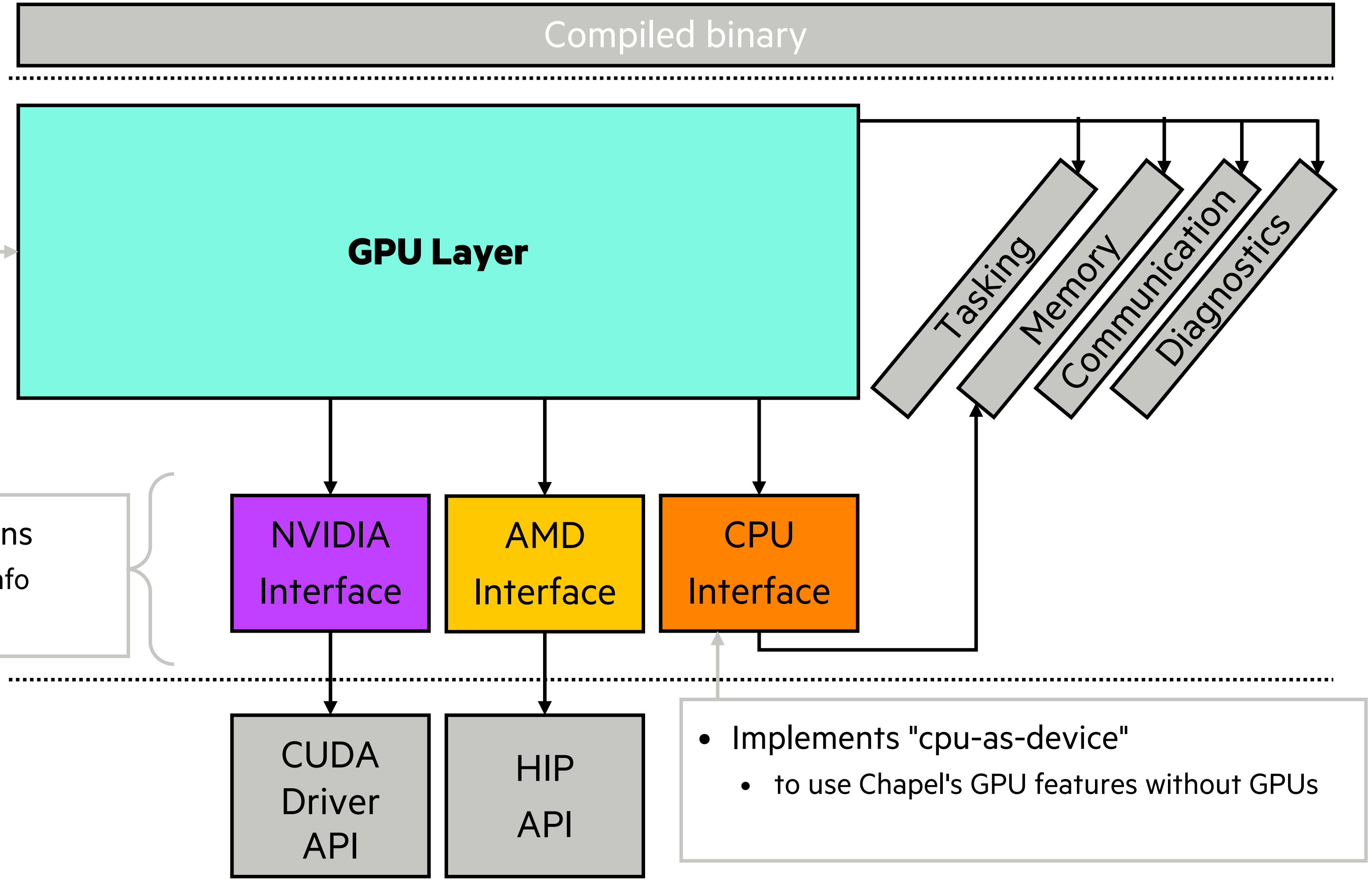
RUNTIME ARCHITECTURE

Interface for:

- Compiler-injected calls
  - e.g. kernel prep and launch
- Extern calls from modules
  - e.g. memory management, data movement

Interacts with the rest of the runtime to:

- Maintain task-private data
  - e.g. GPU streams
- Make host-based allocations
- Move data across locales
- Trigger diagnostics



Compiled binary

GPU Layer

Tasking

Memory

Communication

Diagnostics

NVIDIA Interface

AMD Interface

CPU Interface

CUDA Driver API

HIP API

- Thin layer for primitive GPU operations
  - e.g. call a kernel, initialize driver, query info
- Wraps around drivers

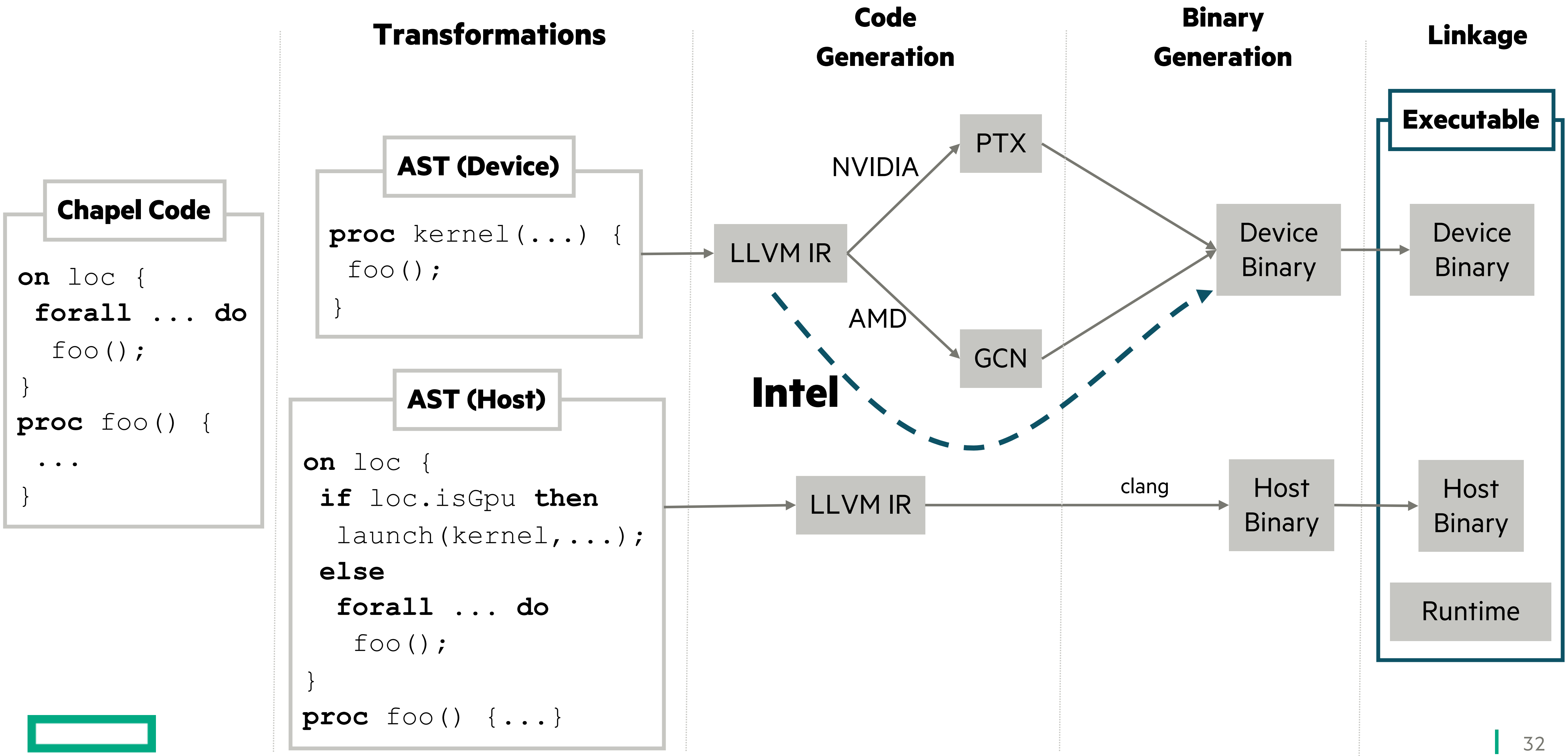
- Implements "cpu-as-device"
  - to use Chapel's GPU features without GPUs

ONGOING WORK AND PLANS

INTEL GPU SUPPORT

- We plan to add support for Intel GPUs
- Status quo for targeting Intel GPUs is using SYCL
  - dpc++ is Intel's fork of LLVM that can target Intel GPUs

COMPILATION TRAJECTORY



Transformations

Code Generation

Binary Generation

Linkage

Chapel Code

```
on loc {
 forall ... do
  foo();
}
proc foo() {
 ...
}
```

AST (Device)

```
proc kernel(...) {
 foo();
}
```

AST (Host)

```
on loc {
 if loc.isGpu then
  launch(kernel,...);
 else
  forall ... do
   foo();
}
proc foo() {...}
```

LLVM IR

NVIDIA

PTX

AMD

GCN

Intel

Device Binary

LLVM IR

clang

Host Binary

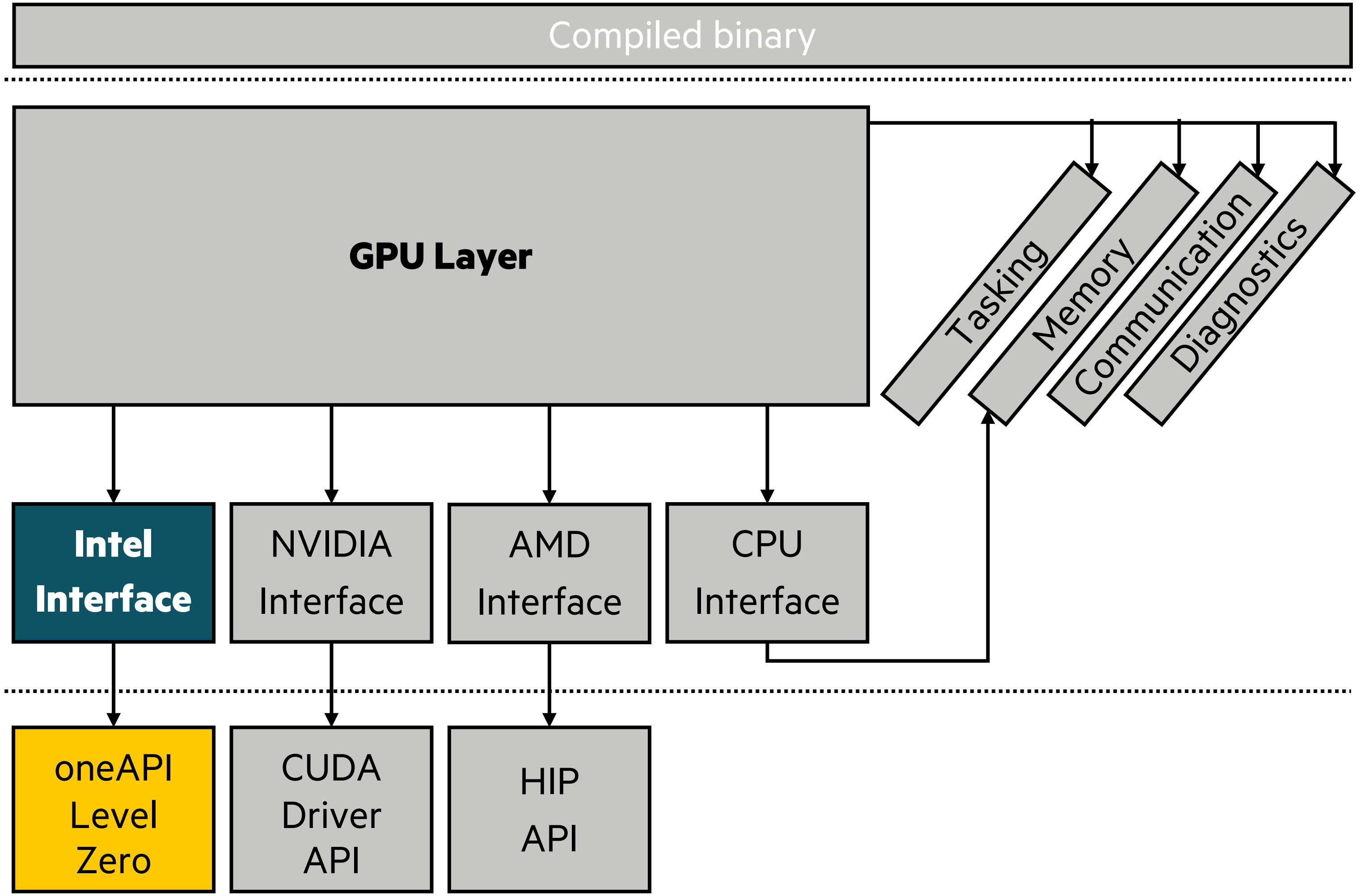
Executable

Device Binary

Host Binary

Runtime

RUNTIME ARCHITECTURE


Compiled binary

GPU Layer

Tasking

Memory

Communication

Diagnostics

Intel Interface

NVIDIA Interface

AMD Interface

CPU Interface

oneAPI Level Zero

CUDA Driver API

HIP API

INTEL GPU SUPPORT

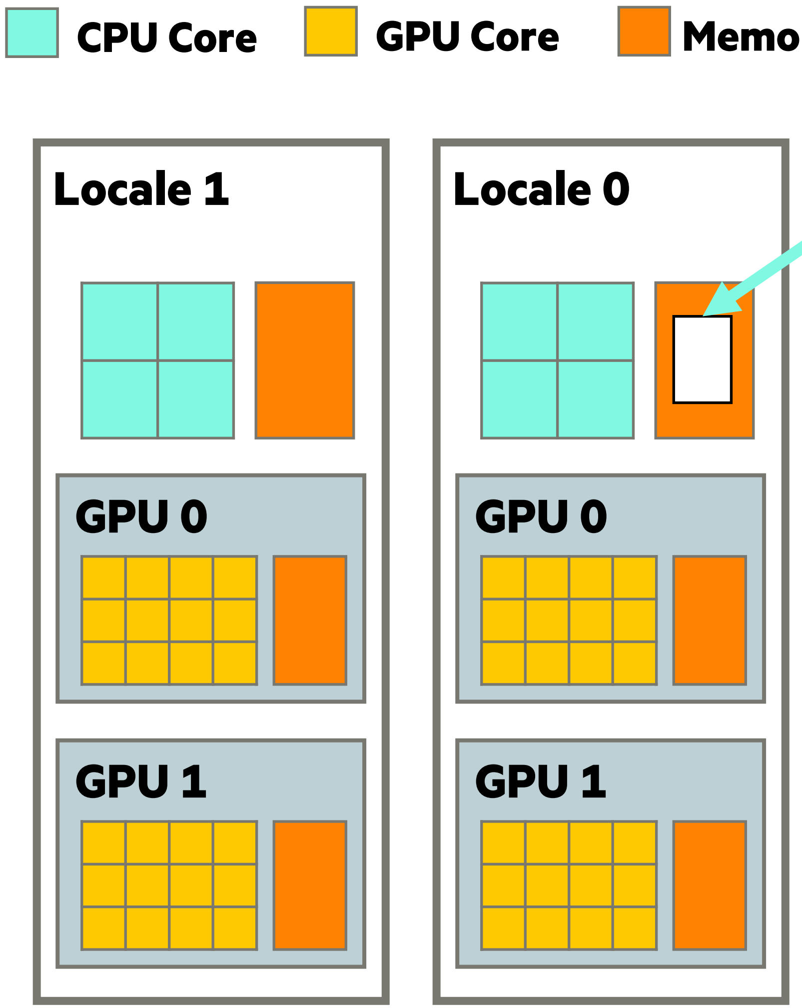
- We plan to add support for Intel GPUs
- Status quo for targeting Intel GPUs is using SYCL
  - dpc++ is Intel's fork of LLVM that can target Intel GPUs

Potential Challenges:

- dpc++ may have diverged from upstream LLVM in other ways, too
  - Using it as our backend is not very straightforward
  - But we have some leads

- We don't foresee any significant challenges on the runtime side at the moment

DISTRIBUTED ARRAY SUPPORT



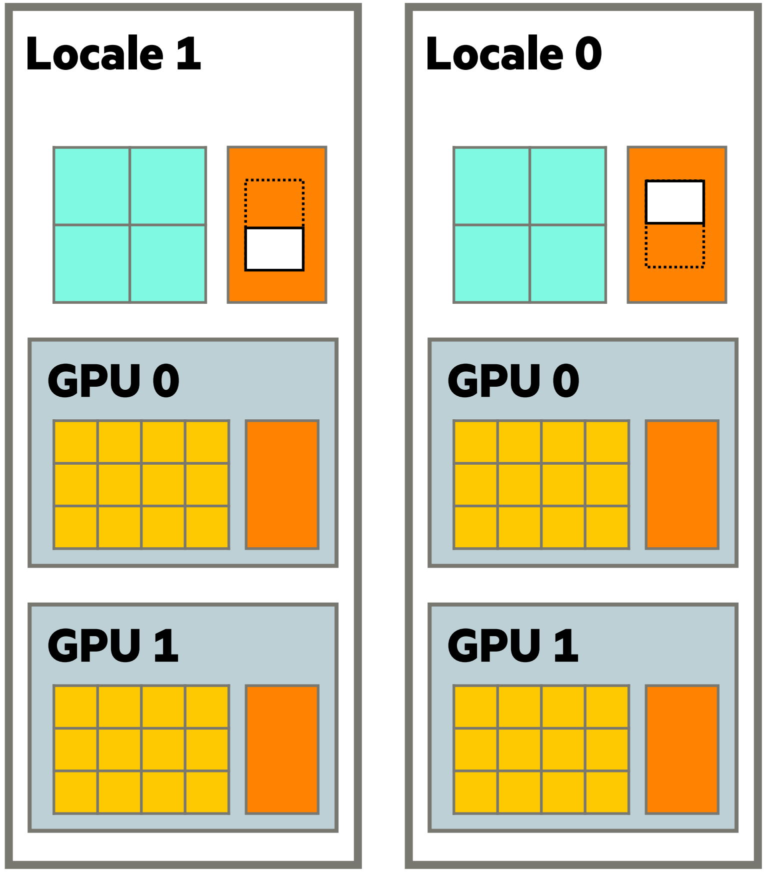
'Arr' is allocated on Locale 0's main memory

```
var Dom = {1..n, 1..m};

var Arr: [Dom] real;


forall a in Arr {
  a = compute(a);
}
```

Executes on Locale 0's CPUs

DISTRIBUTED ARRAY SUPPORT

🟦 CPU Core 🟨 GPU Core 🟧 Memory

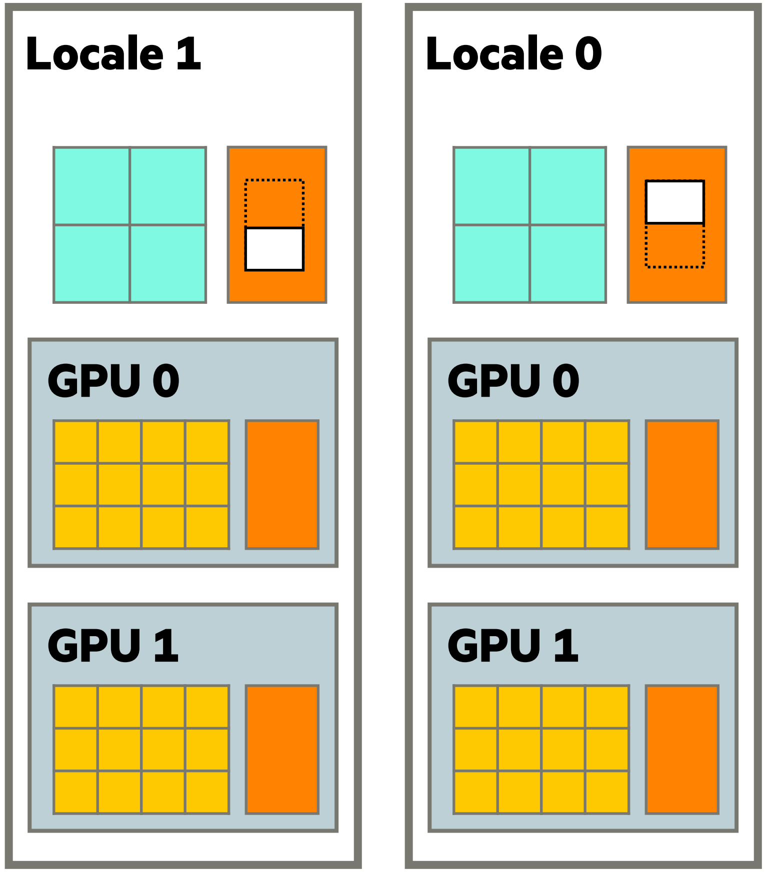


```
var Dom = blockDist.createDomain({1..n, 1..m});

var Arr: [Dom] real;


forall a in Arr {
  a = compute(a);
}
```

Executes on all CPUs

DISTRIBUTED ARRAY SUPPORT

🟦 **CPU Core** 🟨 **GPU Core** 🟧 **Memory**

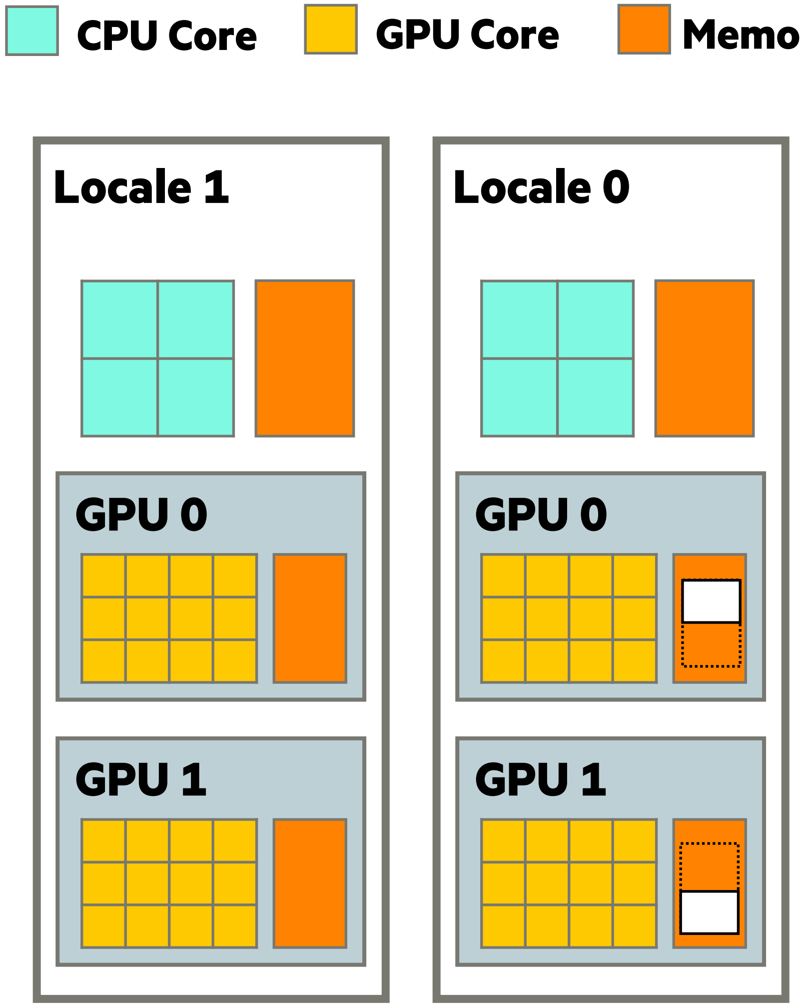


```
var Dom = blockDist.createDomain({1..n, 1..m},
                                 targetLocales=Locales);
var Arr: [Dom] real;


forall a in Arr {
  a = compute(a);
}
```

Redundant: 'Locales' is the default value

Executes on all CPUs

DISTRIBUTED ARRAY SUPPORT

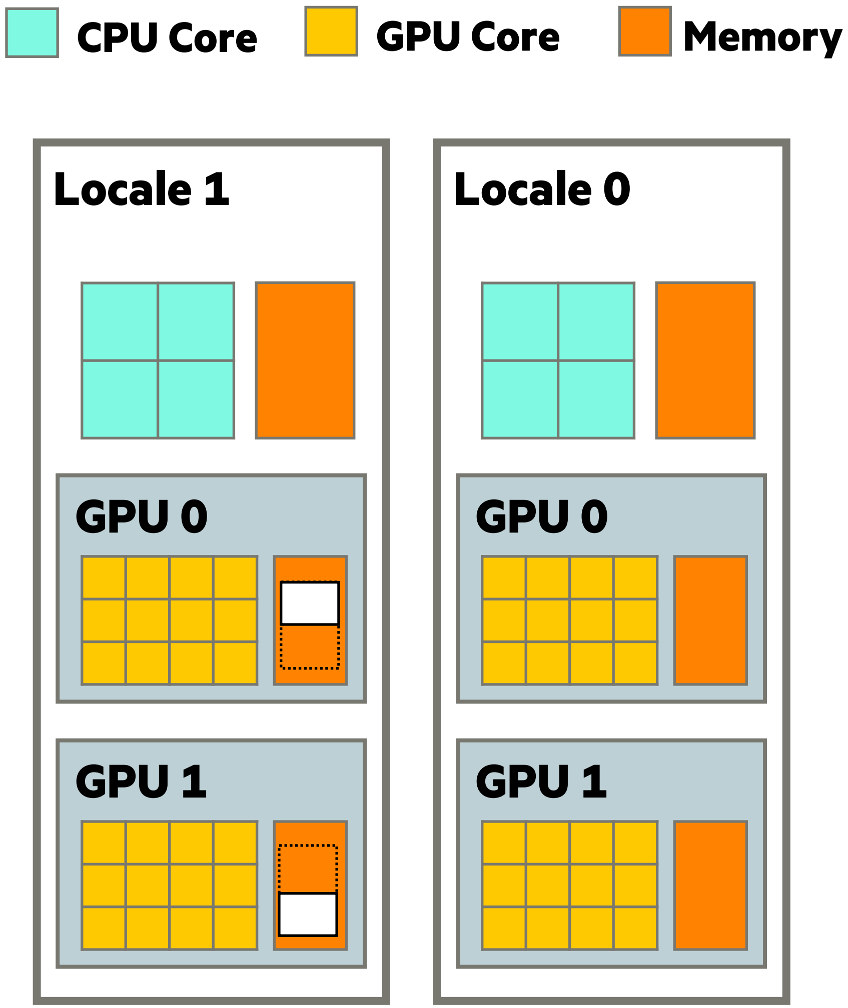


Using 'targetLocales' to distribute arrays on GPU memory is a work-in-progress.

```
var Dom = blockDist.createDomain({1..n, 1..m},
                                 targetLocales=here.gpus);
var Arr: [Dom] real;


forall a in Arr {
  a = compute(a);
}
```

Executes on Locale 0's GPUs

DISTRIBUTED ARRAY SUPPORT

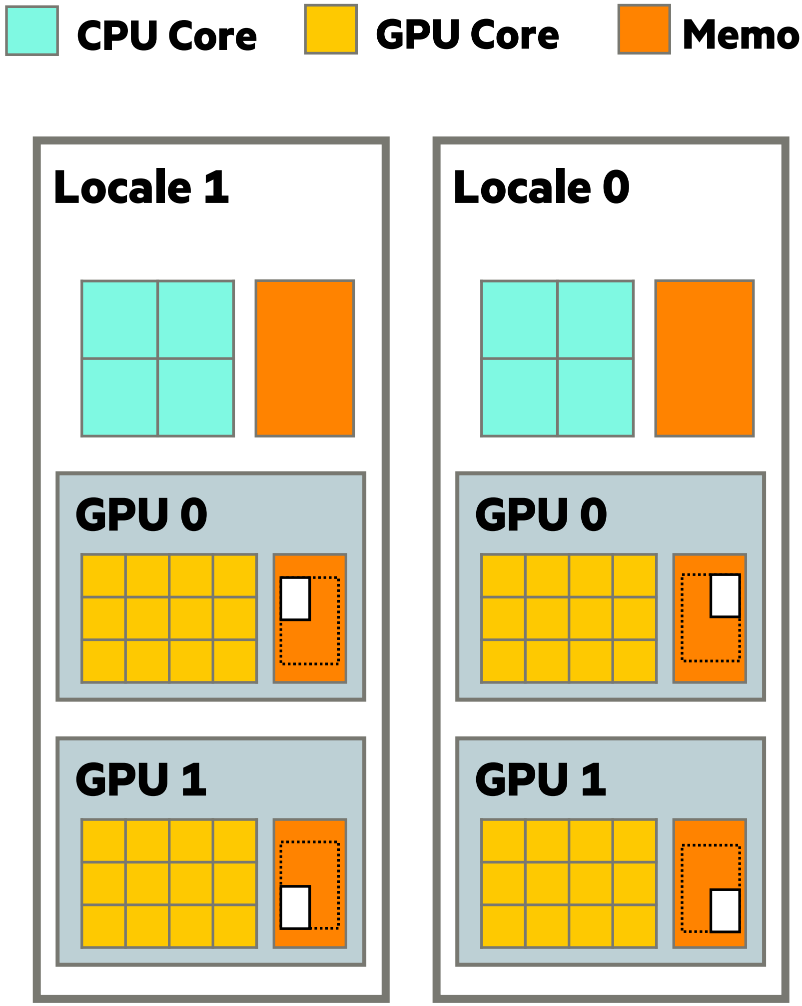


Using 'targetLocales' to distribute arrays on GPU memory is a work-in-progress.

```
var Dom = blockDist.createDomain({1..n, 1..m},
                                 targetLocales=Locales[1].gpus);
var Arr: [Dom] real;


forall a in Arr {
  a = compute(a);
}
```

Executes on Locale 1's GPUs

DISTRIBUTED ARRAY SUPPORT

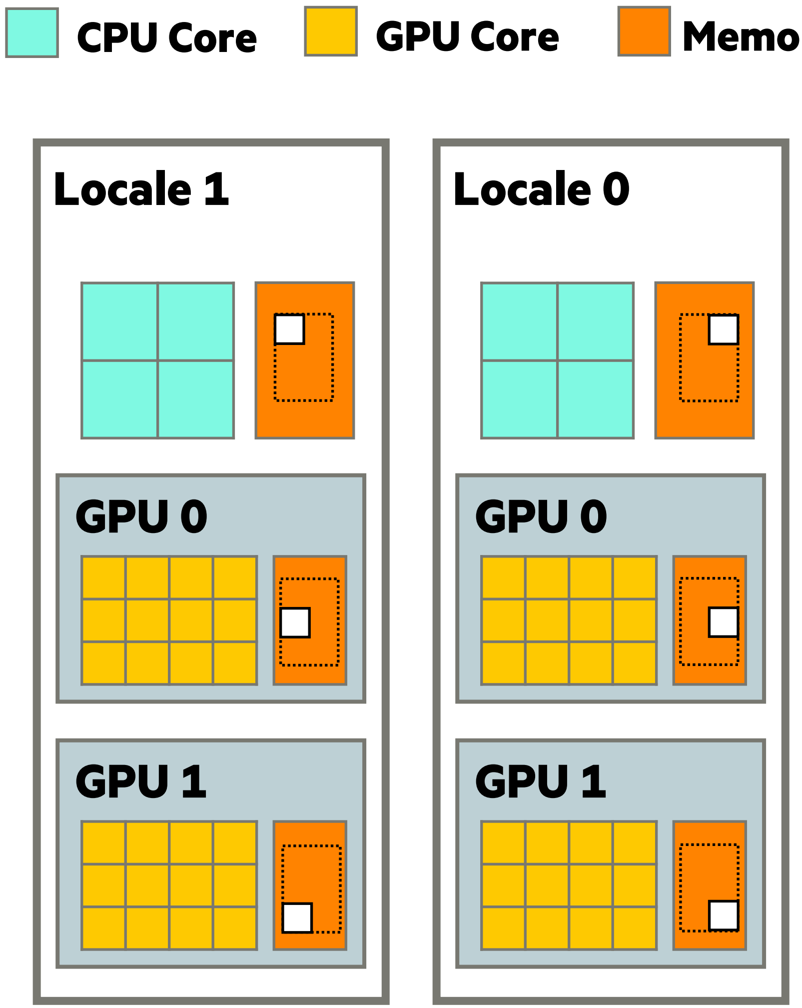


Using 'targetLocales' to distribute arrays on GPU memory is a work-in-progress.

```
var Dom = blockDist.createDomain({1..n, 1..m},
                                 targetLocales=allGpus());
var Arr: [Dom] real;


forall a in Arr {
  a = compute(a);
}
```

Executes on all GPUs

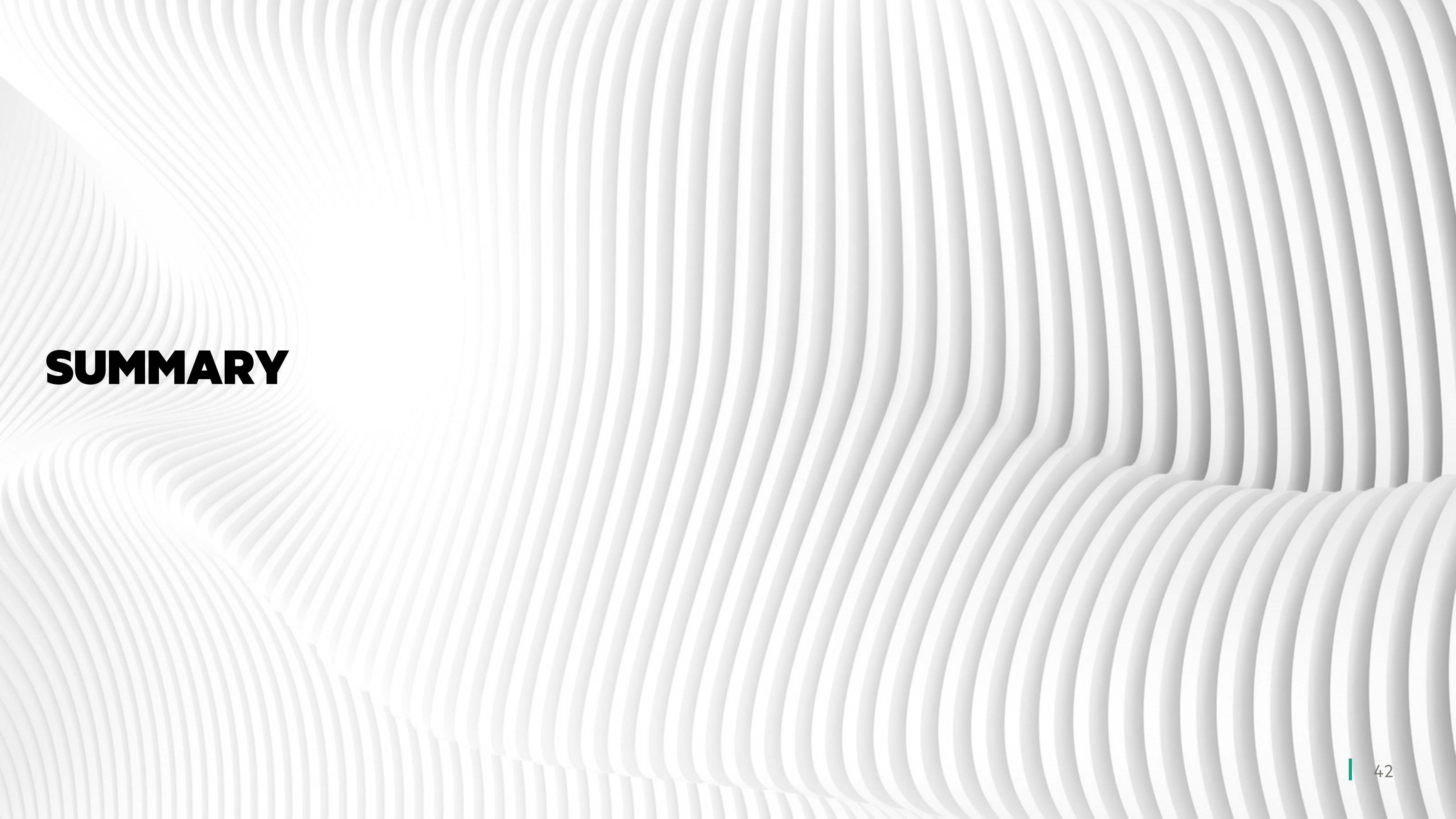
DISTRIBUTED ARRAY SUPPORT



Using 'targetLocales' to distribute arrays on GPU memory is a work-in-progress.

```
var Dom = blockDist.createDomain({1..n, 1..m},
                                 targetLocales=everywhere());
var Arr: [Dom] real;


forall a in Arr {
  a = compute(a);
}
```

Executes on all CPUs and GPUs



SUMMARY

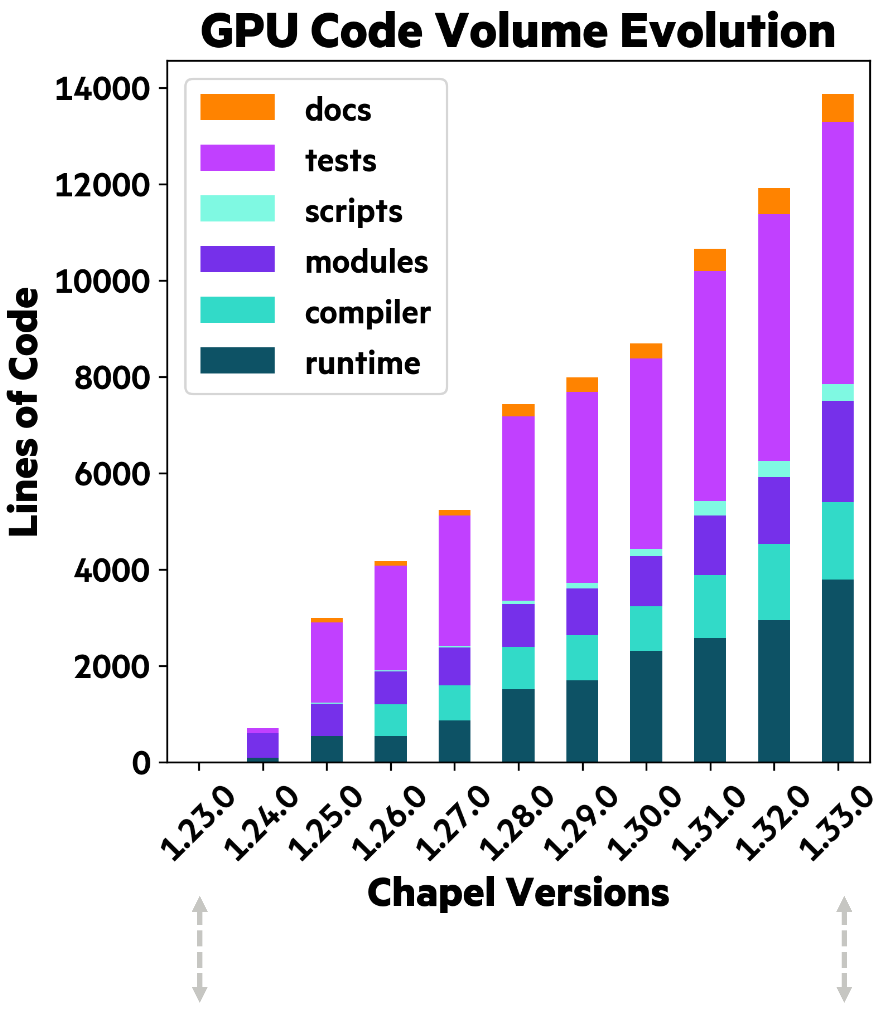
WHERE WE ARE TODAY

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

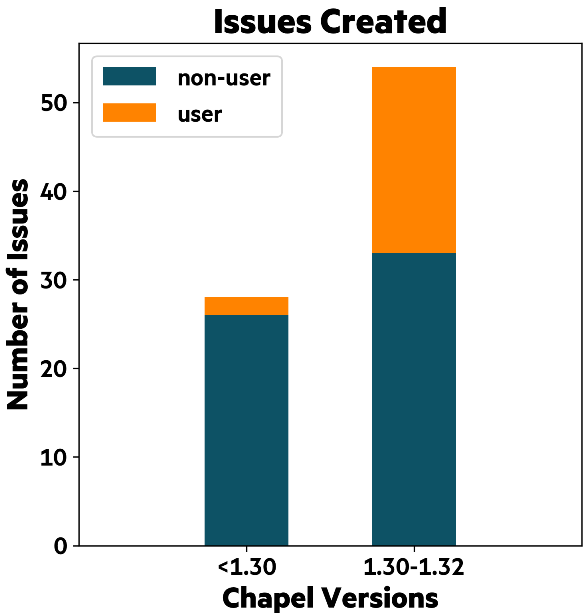


October
2020

December
2023

COMMUNITY REACTION SO FAR

- Ongoing efforts to port existing Chapel applications
- More interactions on our community channels, including GitHub
  - Many new names, too!
- Active collaborations with existing users and researchers

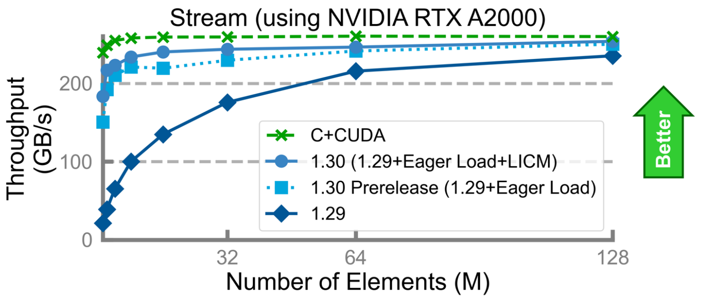


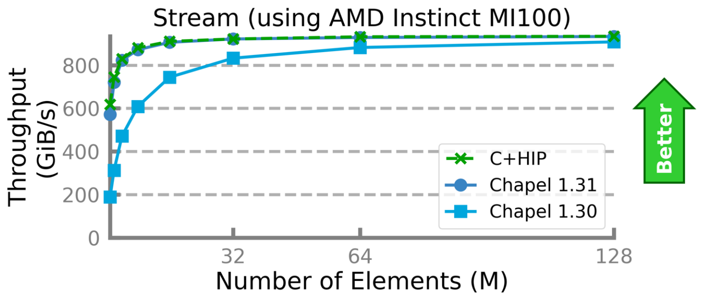
PERFORMANCE STATUS

- We have recently started focusing on performance

HPCC-Stream Performance

- On par with HIP, very close to CUDA





Initial Runs on Frontier

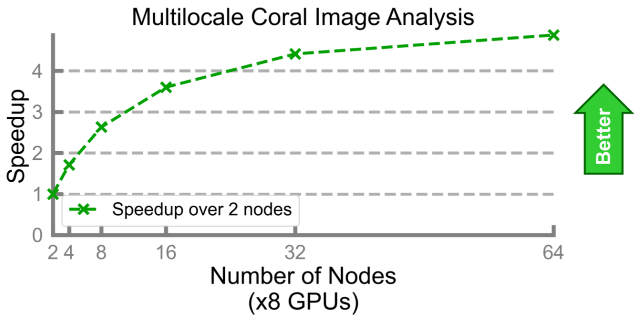
- >10TB/s Stream BW in one node
- ~160GiB/s peer-to-peer BW

User Applications

N-Queens Performance with ChOp

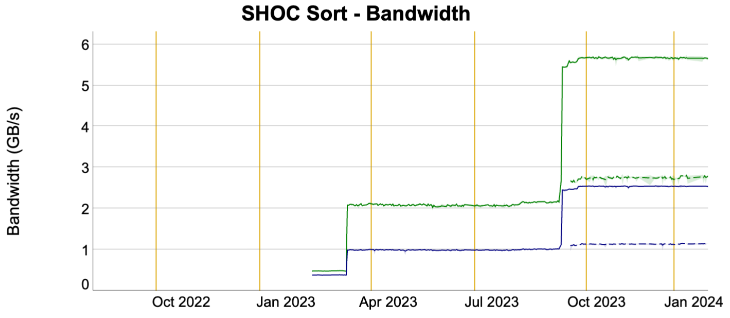
(1x NVIDIA P100)

N	Interop (s)	Native (s)	Off by
15	0.30	0.36	*19%*
16	1.79	2.06	*15%*
17	12.47	14.76	*18%*
18	94.94	110.98	*17%*

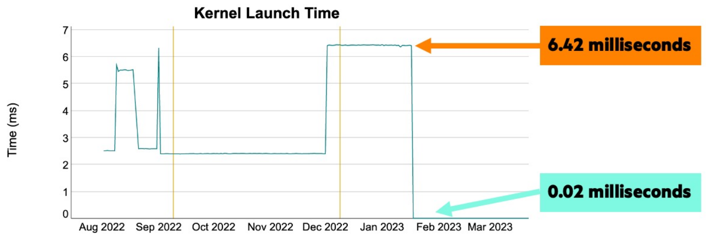


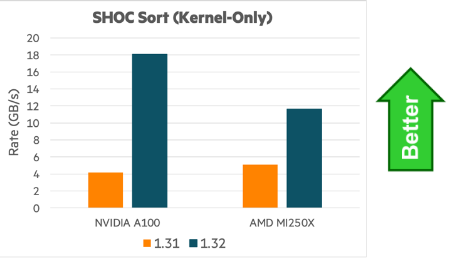
Nightly Performance Testing

- Testing performance on NVIDIA and AMD
  - ~16 tests and counting



Improvement in Each Release

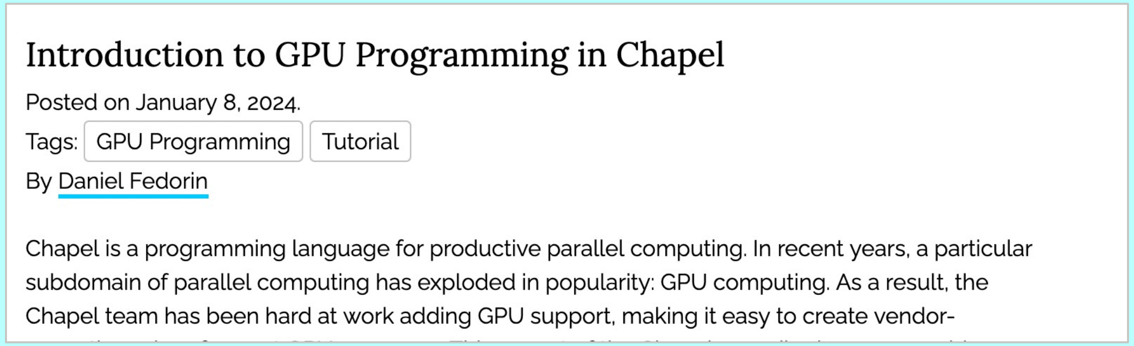




IF YOU WANT TO LEARN MORE ABOUT GPU PROGRAMMING IN CHAPEL

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

- Tutorial on GPU programming in Chapel
  - Covers the basics, more to come soon!



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-

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

- **CHIUW '23 Talk:** updates from May '22-May '23 period
  - https://chapel-lang.org/CHIUW/2023/KayrakliogluSlides.pdf
- **SIAM PP '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

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

chapel-lang.org

CHAPEL RESOURCES

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

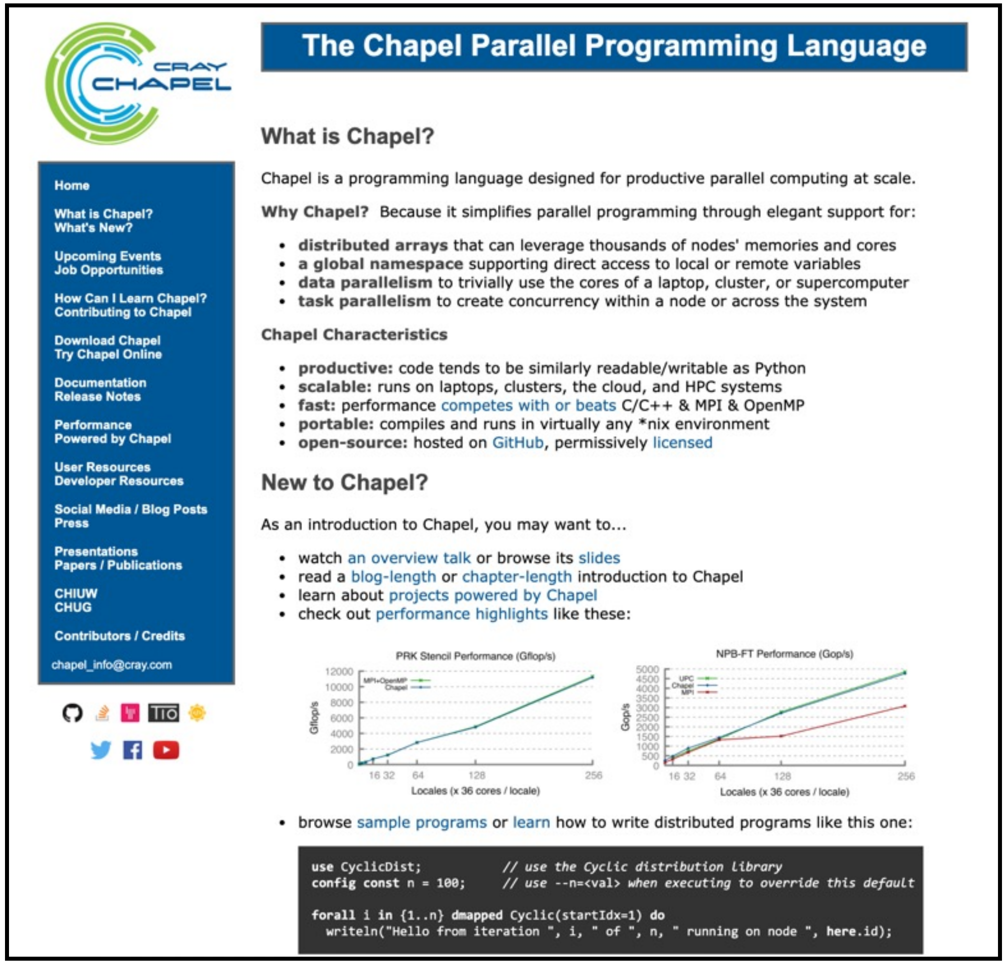
- (points to all other resources)

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



The Chapel Parallel Programming Language

Home

What is Chapel?
What's New?

Upcoming Events
Job Opportunities

How Can I Learn Chapel?
Contributing to Chapel

Download Chapel
Try Chapel Online

Documentation
Release Notes

Performance
Powered by Chapel

User Resources
Developer Resources

Social Media / Blog Posts
Press

Presentations
Papers / Publications

CHIUW
CHUG

Contributors / Credits

chapel_info@cray.com

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:

- **distributed arrays** that can leverage thousands of nodes' memories and cores
- **a global namespace** supporting direct access to local or remote variables
- **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

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 C/C++ & MPI & OpenMP
- **portable:** compiles and runs in virtually any *nix environment
- **open-source:** hosted on GitHub, permissively licensed

New to Chapel?

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

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

- browse sample programs or learn how to write distributed programs like this one:

```
use CyclicDist;           // use the Cyclic distribution library
config const n = 100;     // use --n=<val> when executing to override this default

forall i in {1..n} dmapped Cyclic(startIdx=1) do
  writeln("Hello from iteration ", i, " of ", n, " running on node ", here.id);
```