HIGH-LEVEL, VENDOR-NEUTRAL GPU PROGRAMMING USING CHAPEL

Engin Kayraklioglu
January 9th, 2024
IT IS HARD TO AVOID GPUs IN HPC

With GPUs

Without GPUs

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

www.top500.org
... 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

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
- ...

Can we expand this list to GPUs from all vendors? While using the same expressive features?
APPLICATIONS OF CHAPEL

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
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

Nelson Luis Dias
The Federal University of Paraná, Brazil

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

Nelson Luis Dias
The Federal University of Paraná, Brazil

YOUR APPLICATION HERE?

Active GPU efforts

CHGL: Chapel Hypergraph Library
Louis Jenkins, Cliff Joslyn, Jesun Firoz, et al.
PNNL
1. Read in a $(M \times N)$ raster image of habitat data

2. Create a $(P \times P)$ 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:*
- $M, N > 10,000$
- $P \sim 400$
proc convolve(InputArr, OutputArr) { // 3D Input, 2D Output
    for ... {
        tonOfMath();
    }
}

proc main() {
    var InputArr: ...
    var OutputArr: ...

    convolve(InputArr, OutputArr);
}
CORAL REEF SPECTRAL BIODIVERSITY

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

proc main() {
    var InputArr: ...;
    var OutputArr: ...;

    coforall loc in Locales do on loc {
        coforall gpu in here.gpus do on gpu {
            coforall task in 0..#numWorkers {
                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.

// use all nodes in parallel...
// using GPUs on this node in parallel...
// using numWorkers on this GPU in parallel.

High-level, intuitive array operations work across nodes and/or devices
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 { // use all GPUs on this node in parallel...
        coforall task in 0..#numWorkers { // using parallel tasks on this GPU.
            var MyInputArr = InputArr[...];
            var MyOutputArr: ...;
            convolve(MyInputArr, OutputArr);
            OutputArr[...] = MyOutputArr;
        }
    }
}

CORAL REEF SPECTRAL BIODIVERSITY

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
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.
LOCASES 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
KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

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

Locale 0  Locale 1  Locale 2  Locale 3

Processor Core
Memory
PARALLELISM AND LOCALITY

```
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

\[
\text{Locale 0} \\
\text{var A: [1..2, 1..2] real;}
\]

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

\[
\text{writeln(A);}
\]
PARALLELISM AND LOCALITY

The coforall loop creates a parallel task per iteration

```plaintext
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);
```
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

<table>
<thead>
<tr>
<th>Locale 0</th>
<th>Locale 1</th>
<th>Locale 2</th>
<th>Locale 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="GPU_0" alt="" /></td>
<td><img src="GPU_1" alt="" /></td>
<td><img src="GPU_0" alt="" /></td>
<td><img src="GPU_1" alt="" /></td>
</tr>
<tr>
<td><img src="GPU_2" alt="" /></td>
<td><img src="GPU_3" alt="" /></td>
<td><img src="GPU_2" alt="" /></td>
<td><img src="GPU_3" alt="" /></td>
</tr>
<tr>
<td><img src="CPU_Core" alt="" /></td>
<td><img src="CPU_Core" alt="" /></td>
<td><img src="CPU_Core" alt="" /></td>
<td><img src="CPU_Core" alt="" /></td>
</tr>
<tr>
<td><img src="Memory" alt="" /></td>
<td><img src="Memory" alt="" /></td>
<td><img src="Memory" alt="" /></td>
<td><img src="Memory" alt="" /></td>
</tr>
</tbody>
</table>

**Legend:**
- *GPU Core*
- *CPU Core*
- *Memory*
PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUs

```
locale 0

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

```pascal
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

```plaintext
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

```plaintext
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?
**LOOP BODY IS OUTLINED**

**AST (Device)**

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

**AST (Host)**

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

**Runtime's kernel launcher is called**

**Code Generation**

- **LLVM IR**
  - **NVIDIA**: PTX
  - **AMD**: GCN

**Binary Generation**

- **Device Binary**
- **Host Binary**

**Linkage**

- **Executable**
  - **Device Binary**
  - **Host Binary**

**Runtime**

- **clang**
- **ptxas, fatbinary**
- **llvm-mc, clang tools**

**Transformations**

- **Chapel Code**
  - `on loc {
    forall ... do
      foo();
  }
proc foo() { ...}
`
**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

- 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

**Chapel Code**

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

**AST (Device)**

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

**AST (Host)**

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

Compiled binary

- GPU Layer
  - Intel Interface
    - oneAPI Level Zero
  - NVIDIA Interface
    - CUDA Driver API
  - AMD Interface
    - HIP API
  - CPU Interface
- Tasking
- Memory
- Communication
- Diagnostics
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

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

var Arr: [Dom] real;

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

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

Executes on Locale 0's CPUs
var Dom = blockDist.createDomain({1..n, 1..m});

var Arr: [Dom] real;

forall a in Arr {
    a = compute(a);
}
DISTRIBUTED ARRAY SUPPORT

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

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

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

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

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

var Arr: [Dom] real;

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

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

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

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

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

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

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

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

![GPU Code Volume Evolution](chart.png)

<table>
<thead>
<tr>
<th>Chapel Versions</th>
<th>October 2020</th>
<th>December 2023</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.23.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.24.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.25.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.26.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.27.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.28.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.29.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.30.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.31.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.32.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.33.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lines of Code
- docs
- tests
- scripts
- modules
- compiler
- runtime
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

User Applications

N-Queens Performance with ChOp
(1x NVIDIA P100)

<table>
<thead>
<tr>
<th>N</th>
<th>Interop (s)</th>
<th>Native (s)</th>
<th>Off by</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>0.30</td>
<td>0.36</td>
<td>19%</td>
</tr>
<tr>
<td>16</td>
<td>1.79</td>
<td>2.06</td>
<td>15%</td>
</tr>
<tr>
<td>17</td>
<td>12.47</td>
<td>14.76</td>
<td>18%</td>
</tr>
<tr>
<td>18</td>
<td>94.94</td>
<td>110.98</td>
<td>17%</td>
</tr>
</tbody>
</table>

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

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

Improvement in Each Release
- 442 milliseconds
- 0.02 milliseconds
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!

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
  - SIAM PP ’22 Talk: a lot of details on how the Chapel compiler works to create GPU kernels
  - Recent Release Notes: almost everything that happened in each release
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