Hewlett Packard
Enterprise

CHAPEL 1.31/1.32 RELEASE NOTES: GPU SUPPORT

Chapel Team
June 22, 2023 / September 28, 2023

GPU SUPPORT OUTLINE

- Background
- Features
- Portability
- Performance
- Next Steps

BACKGROUND

GPU SUPPORT

Background

- We are adding native GPU support to Chapel
  - A highly desired feature, given the potential to be a clean and portable way of programming GPUs
  - GPUs are more and more common in supercomputers
    - Over 95% of the compute capability on Frontier (currently #1 on the top-500) comes from its GPUs
- In earlier releases, we've…

  …moved from an idea (**1.23**), to a demo (**1.24**), to a user-accessible feature on NVIDIA GPUs (**1.25**), …

  …to being able to drive multiple GPUs on one locale (**1.26**), and then multiple locales (**1.27**).
- We started to focus on performance and portability during **1.29** / **1.30**

- **1.31** / **1.32**: continued push on performance and portability, responded to uptick in user requests
  - **Performance:** optimizations impacting many benchmarks, ability to use Chapel tasks with GPUs
  - **Portability:** AMD/NVIDIA parity, initial support for CUDA 12/ROCm 5, new cpu-as-device mode
  - **Community:** new users trying out GPU support, significant increase in GitHub interactions
  - Also new features for users and capabilities for developers

GPU SUPPORT

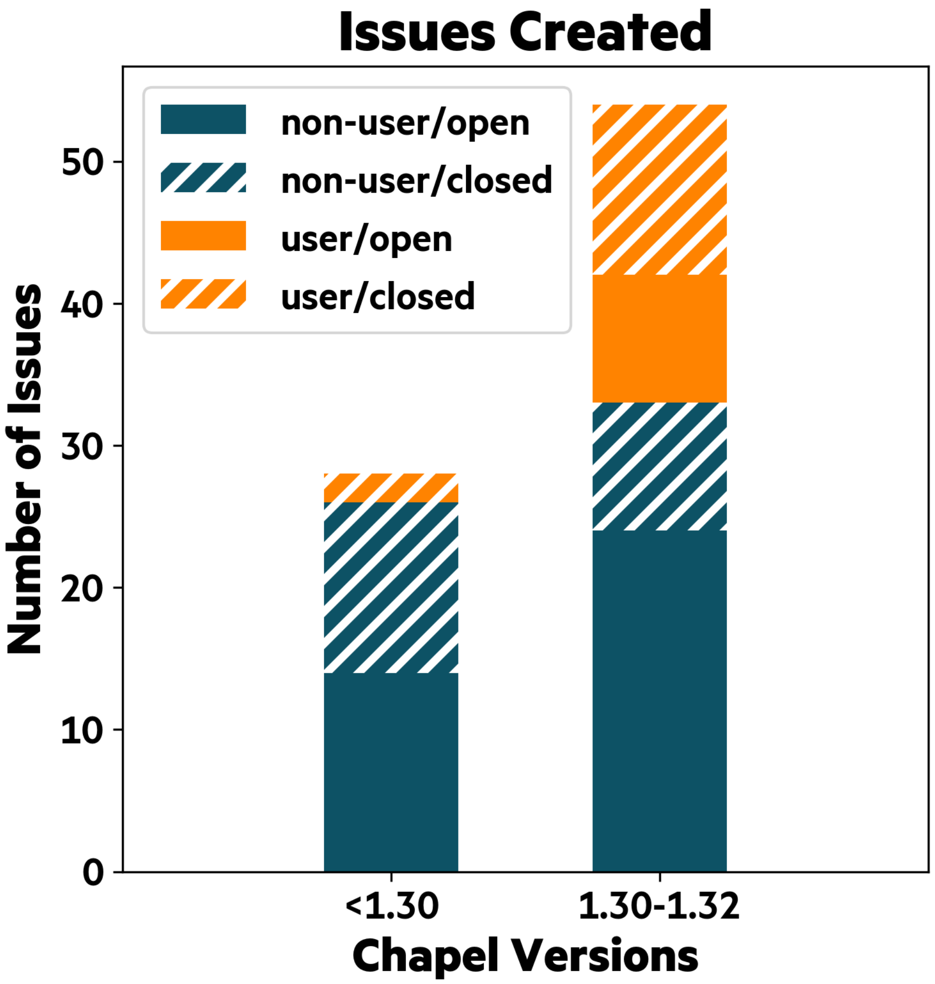
GitHub Activity Summary

- GPU support has started to receive attention

- Before 1.30:
  - **2 user-reported issues** were opened
- Between 1.30 and 1.32:
  - we had **21 user-reported issues**

- During 1.31/1.32, we prioritized resolving user issues
  - we closed **27 total issues**,
  - **14** of them were reported by users

- We also started to report issues publicly ourselves
  - … while migrating internal discussions to the public repo



CRASH COURSE IN GPU PROGRAMMING USING CHAPEL

GPU SUPPORT

Vector Increment Example: Basics

```
on here.gpus[0] {



  var GpuVec: [1..n] int;
  GpuVec += 1;
  writeln(GpuVec);

}
```

'on' statement targets a GPU

array data will be allocated on the targeted GPU

data-parallel operations will launch as a GPU kernel

GPU SUPPORT

Vector Increment Example: Data Offload via Bulk Array Assignment

```
var CpuVec: [1..n] int;

  on here.gpus[0] {




      var GpuVec = CpuVec;
      GpuVec += 1;
      CpuVec = GpuVec;


  }

writeln(CpuVec);
```

host-to-device copy

device-to-host copy

GPU SUPPORT

Vector Increment Example: Multiple GPUs via 'coforall'

```
var CpuVec: [1..n] int;

  coforall gpu in here.gpus do on gpu {


      const myChunk = ...;

      var GpuVec = CpuVec[myChunk];
      GpuVec += 1;
      CpuVec[myChunk] = GpuVec;


  }

writeln(CpuVec);
```

'coforall' creates a task per local GPU

a slice of the data is copied between host and device

GPU SUPPORT

Vector Increment Example: Multiple GPUs on Multiple Locales

```
var CpuVec: [1..n] int;
coforall loc in Locales do on loc {
  coforall gpu in here.gpus do on gpu {


      const myChunk = ...;

      var GpuVec = CpuVec[myChunk];
      GpuVec += 1;
      CpuVec[myChunk] = GpuVec;


  }
}
writeln(CpuVec);
```

'coforall' over all locales

GPU SUPPORT

Vector Increment Example: Multiple GPUs using Multiple Tasks on Multiple Locales

```
var CpuVec: [1..n] int;
coforall loc in Locales do on loc {
  coforall gpu in here.gpus do on gpu {
    coforall workerId in 0..<numTasks {

      const myChunk = ...;

      var GpuVec = CpuVec[myChunk];
      GpuVec += 1;
      CpuVec[myChunk] = GpuVec;

    }
  }
}
writeln(CpuVec);
```

'coforall' to create multiple tasks per GPU

This pattern has significantly improved performance in 1.32

See the "Performance" part of this deck.

GPU SUPPORT

Overview of Changes in 1.31 and 1.32

Performance:

- Faster array access in kernels
- Faster Math library calls
- Faster multitasking on GPUs
- Turned the faster memory strategy on by default
- Peer-to-peer access features and exploration

Portability:

- CPU-as-Device mode
- AMD/NVIDIA feature and performance parity
- Initial Intel exploration
- CUDA 12/ROCm 5 support

New Features and Capabilities:

- Standalone atomic functions
- '--report-gpu' compiler flag
- Ability to compile for multiple NVIDIA architectures
- Improved debugging features:
  - Ability to inspect assembly for AMD GPUs
  - Improved auto-generated kernels' names
  - New loop attribute '@assertOnGpu'

FEATURES

- Atomic Operations
- '--report-gpu' flag
- Assembly Inspection
- '@assertOnGpu' attribute
- Multi-arch compilation
- Improved Kernel Naming

ATOMIC OPERATIONS ON GPU

Background: GPUs have support for atomic operations (add, compare-and-swap, etc.)

This Effort: Added the following procedures for atomic operations to the GPU module:

- `gpuAtomicAdd`
- `gpuAtomicSub`
- `gpuAtomicExch`
- `gpuAtomicMin`
- `gpuAtomicMax`
- `gpuAtomicInc`
- `gpuAtomicDec`
- `gpuAtomicAnd`
- `gpuAtomicOr`
- `gpuAtomicXor`
- `gpuAtomicCAS`

Status: Almost all operations are supported on NVIDIA and AMD GPUs

- Caveat: 64-bit, signed, atomic 'min' and 'max' operations do not work when compiling for AMD
  - These operations are not supported in HIP version < 5.7 (we currently support 4.0–5.4)
  - We produce a compile-time error if these are used and 'CHPL_GPU=amd' is set

Next Steps:

- Allow using variables with Chapel's 'atomic' type and have them lower to these calls as appropriate (#23619)
- Enable atomic min and max on AMD GPUs once we support HIP versions >= 5.7

--REPORT-GPU FLAG

Background: Chapel generates kernels for all GPU-eligible loops

- Users may want to know what loops are and are not GPU-eligible
- 'assertOnGpu' does a compile-time eligibility check, but needs to be applied manually to all loops

This Effort: Added '--report-gpu' to chpl to dump loop eligibility information

- We report on all loops that are order-independent and not already in a GPU kernel

Impact: The following code produces the following output when compiled with '--report-gpu':

```
foreach i in 0..10 do A[i] = callToExtern();
foreach i in 0..10 do A[i] *= 2;
foreach i in 0..10 do A[i] += 2;
```

```
GPU INELIGIBLE LOOPS:
---------------------
foo.chpl:1

GPU ELIGIBLE LOOPS:
-------------------
foo.chpl:2
foo.chpl:3
```

Next Steps: Consider increasing the verbosity for when we report GPU-ineligibility ([#23620](#23620))

EMITTING GPU ASSEMBLY WITH --SAVEC FLAG

Background: --savec dumps code that can help users gain performance insights

- When using the C backend, it saves C files
- When using LLVM, it saves various llvm-related intermediate files
  - (the name "savec" needs to change, at least for LLVM, see #18602)
- When compiling for NVIDIA GPUs, it also stores a PTX assembly file
  - but previously we did not do this for AMD

This Effort: Ensured --savec outputs GPU-related assembly for NVIDIA and AMD

Impact:

- Regardless of GPU target, we output a 'chpl__gpu.s'
- In the generated assembly, kernels are named 'chpl_gpu_kernel_<fileName>_line_<num>'
- We now documented this in the technote and intend to support it going forward

ASSERT-ON-GPU ATTRIBUTE

Background:

- Ensuring that loops were GPU-eligible was handled by a special 'assertOnGpu( )' function
- Calls to 'assertOnGpu( )' were either compile-time or run-time depending on its position, which was unusual
  - If 'assertOnGpu( )' was a top-level statement in an ineligible loop, compiler reported an error immediately

This Effort:

- Use recently-added loop attributes to introduce '@assertOnGpu', which always performs a compile-time check
- Precludes the need for differentiating function behavior *if it's at the top level*

```
@assertOnGpu
foreach a in A do a += 1;
```

Status:

- '@assertOnGpu' is the preferred way to check GPU eligibility
  - the standalone 'assertOnGpu' function is deprecated

Next Steps:

- Investigate if a runtime-only assertion (like 'assertOnGpu()' not-at-top-level) is necessary

MULTI-ARCHITECTURE GPU EXECUTABLES

Background:

- It’s common for GPU-enabled programs to embed multiple GPU binaries for different architectures
  - Enables a compiled program to run on devices with different GPU hardware
  - e.g., a cluster with different GPU nodes, or a laptop with dedicated and integrated GPU

This Effort:

- Added prototypical support for multi-architecture executables to Chapel’s GPU functionality

Status:

- Initial support for multi-architecture executables for NVIDIA
  - To access, pass comma-separated architectures to ‘--gpu-arch’

```
> chpl --gpu-arch sm_70,sm_80
```

- Current approach relies on using the lowest-common version of PTX for named architectures
  - with additional effort, could specialize PTX per architecture

Next Steps:

- Investigate additional specialization for architectures and multi-vendor support (#22783)

GPU KERNEL NAMING

Background:

- Chapel generates GPU kernels by translating loops into procedures (named 'chpl_gpu_kernel')
- If multiple kernels are present, the built-in mangling appended '_1', '_2', and more
- However, 'chpl_gpu_kernel_1' isn't very descriptive, and doesn't make for easy debugging

This Effort:

- Change the GPU kernel naming policy to include the filename and line number. e.g.,

```
chpl_gpu_kernel_fileName_line_13
chpl_gpu_kernel_fileName_line_37
```

Status:

- Kernel naming changes are available in 1.32

PORTABILITY

- AMD/NVIDIA Parity
- Intel Explorations
- CPU-as-Device mode
- CUDA 12/ROCm 5 support

GPU ARCHITECTURE FEATURE PARITY

Background: In 1.30, some Chapel code was not portable across AMD and NVIDIA GPUs

- Specifically, using the 64-bit versions of these functions caused compile-time failures when building for AMD:

```
acos      acosh     asin      asinh     atan      atan2
atanh     cbrt      cosh      erf       erfc      ldexp
lgamma    log1p     sinh      tan       tanh      tgamma
```

This Effort: Fixed a bug causing us to erroneously link to the wrong version of these math functions

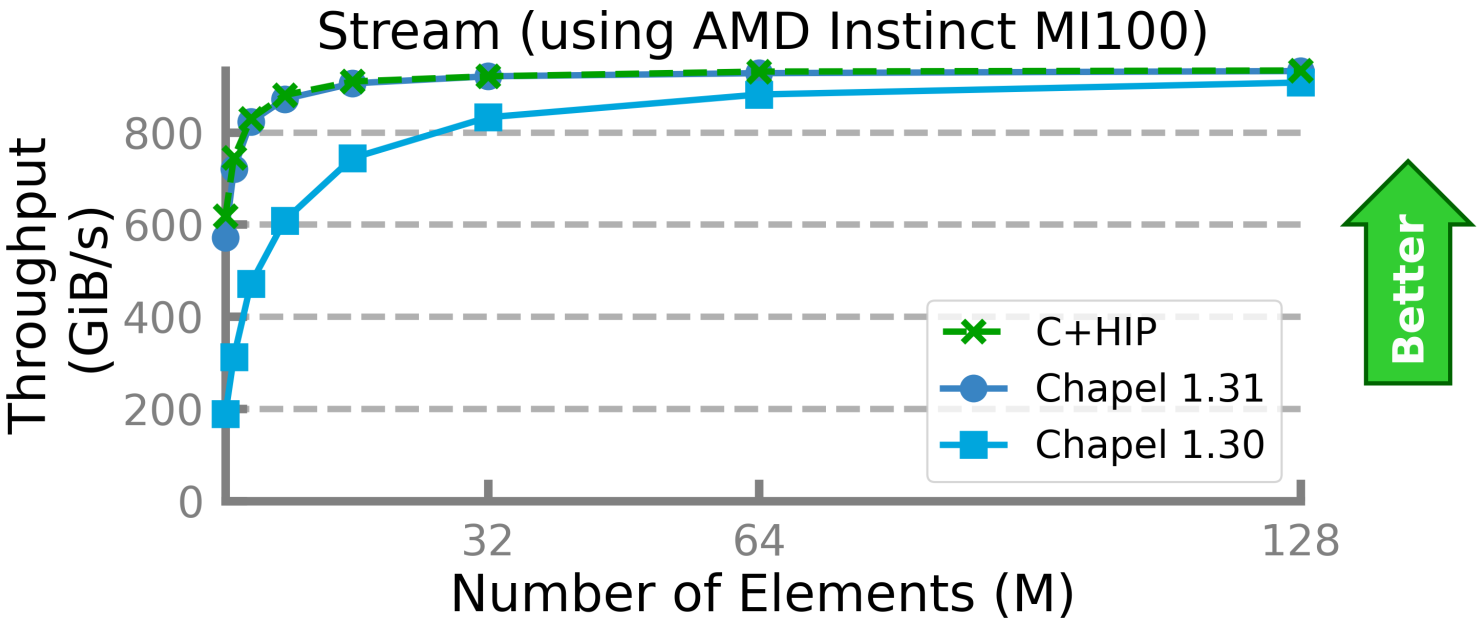
Status: We now support the same math functions for NVIDIA and AMD

GPU ARCHITECTURE PERFORMANCE PARITY

Background: In 1.30, HPCC-Stream was competitive with CUDA on NVIDIA but not with HIP on AMD

This Effort: Updated runtime to avoid calling a deprecated HIP API

Impact: Stream now performs competitively to C+HIP on AMD



TARGETING INTEL GPUS

Background:

- Chapel supports targeting NVIDIA and AMD GPUs; but Intel GPUs are not supported, yet
  - LLVM does not support targeting Intel GPUs

This Effort:

- We investigated Intel's LLVM-based 'dpc++' compiler
  - Discovered that default builds may not be suitable for use as the system LLVM
  - Headers and some tools are missing

Next Steps:

- Allow Chapel to be built with Intel's LLVM as the system LLVM
  - Create documentation for it for advanced users
- Implement a runtime layer for Intel GPUs based on oneAPI Level Zero

CPU-AS-DEVICE MODE

Background and This Effort

Background:

- Chapel's GPU support required the runtime to be built with CUDA or HIP as a dependency
  - This meant that even simple development must be done on a system with actual GPUs
- Being able to start HPC-oriented development on a personal computer is an important part of productivity
  - e.g., Chapel also allows multilocale development on a personal computer

This Effort: Chapel now has a *cpu-as-device* mode for GPU programming without GPUs

- No CUDA/HIP dependencies, no need for actual GPUs
- To enable this mode:

```
> export CHPL_LOCALE_MODEL=gpu    # required for GPU support in general
> export CHPL_GPU=cpu             # mandatory to enable cpu-as-device mode. i.e., will never be set automatically
```

CPU-AS-DEVICE MODE

Status

- Compiler works similarly, but the original loop will always execute
- Runtime's calls bump up diagnostic counters as appropriate, redirect to other parts of the runtime
  - i.e., GpuDiagnostics can be used normally in most cases

```
foreach i in 1..n do
  foo()
```

Translates into

```
if (on_gpu()) {
  launch_kernel(...);
}
foreach i in 1..n do
  foo();
```

Runtime code:

```
void launch_kernel(...) {
  num_launches += 1;
  return;
}
```

- '@assertOnGpu' on a loop generates:
  - Compiler error: if the loop is not GPU-eligible
  - Runtime warning: if the loop is run on a non-GPU locale
    - The warning can be disabled by setting the 'CHPL_GPU_NO_CPU_MODE_WARNING' environment variable

CPU-AS-DEVICE MODE

Next Steps

- We plan to address some behavior differences we observed
  - Nested GPU-eligible loops cause GpuDiagnostics to register more kernel launches than expected
  - Argument passing and outer variable usage details are not captured in this mode
    - We were unable to reproduce some actual GPU bugs in this mode
  - The generated kernel is discarded while generating the final code
    - There's no generated kernel code that is very useful for advanced debugging during development

```
foreach i in 1..n do
  foo()
```

Will translate into

```
if (on_gpu()) {
  launch_kernel(...);
}
else { // need 'else' now
  foreach i in 1..n do
    foo();
}
```

Runtime code:

```
void launch_kernel(...) {
  num_launches += 1;
  for (int threadIdx...
    call_kernel(kernel,
                threadIdx,
                ...);
  return;
}
```

CUDA 12.X SUPPORT

Background:

- Chapel supported CUDA 11.x and 10.x with some limitations
- CUDA 12.x was not supported before
  - Main blocker: LLVM/Clang 15 (highest version Chapel supports) does not support CUDA 12.x
  - Noted by multiple users
- LLVM/Clang 16 supports CUDA 12

This Effort:

- We patched our bundled LLVM (version 15) to support CUDA 12
- Unsupported versions generate an error while building Chapel

Status:

- CUDA 12 is now supported only when using the bundled LLVM

Next Steps:

- Complete LLVM 16 upgrade to enable CUDA 12 support with system LLVM too
- Consider dropping CUDA 10.x support
  - Should be a documentation change only: we do not maintain any code to support 10.x specifically

ROCM 5.X SUPPORT

Background:

- Chapel supported ROCm 4.x
- ROCm 5.x was not tested before

Status:

- Unsupported versions generate an error while building Chapel
- **5.0, 5.1:** Fully supported
- **5.2-5.4:** Supported, but deprecation warnings from clang are expected
  - The way the compiler uses a clang tool to bundle device and host binaries is deprecated
  - We plan to fix this soon
- **5.5+:** Not supported
  - These versions require LLVM 16
  - There may be a way to use LLVM 15, or patch it similarly to LLVM 16
  - For now, we are waiting on the LLVM 16 upgrade
- **5.7+:** Not supported, but required for 64-bit, signed 'gpuAtomicMax' and 'gpuAtomicMin' support
- See #23480 for the most up-to-date status of ROCm 5.x support

PERFORMANCE

- Faster Array Access
- Peer-to-Peer Access
- Array-On-Device
- Task Parallelism with GPUs
- Faster Math Library Calls
- GPU Specialization

FASTER ARRAY ACCESS IN KERNELS

Background and This Effort

Background:

- Arrays have two layers of indirection to get to underlying data
- Loop Invariant Code Motion (LICM) is an optimization that moves code from inside to outside a loop
  - Helps avoid repetitive computations that always have the same value (e.g., 1+1).
  - Can be used to move array metadata access, too
- Chapel’s LICM optimization is conservative; arrays passed by reference are not considered “constant”

```
proc copyArray(ref A: [?D] int, ref B: [D] int) {
  foreach i in A.domain do B[i] = A[i];
}
```

Result:

4 metadata accesses *per iteration!*

This Effort:

- Arrays passed by reference to GPU kernels won’t be changed from outside
  - Relax LICM rules to match

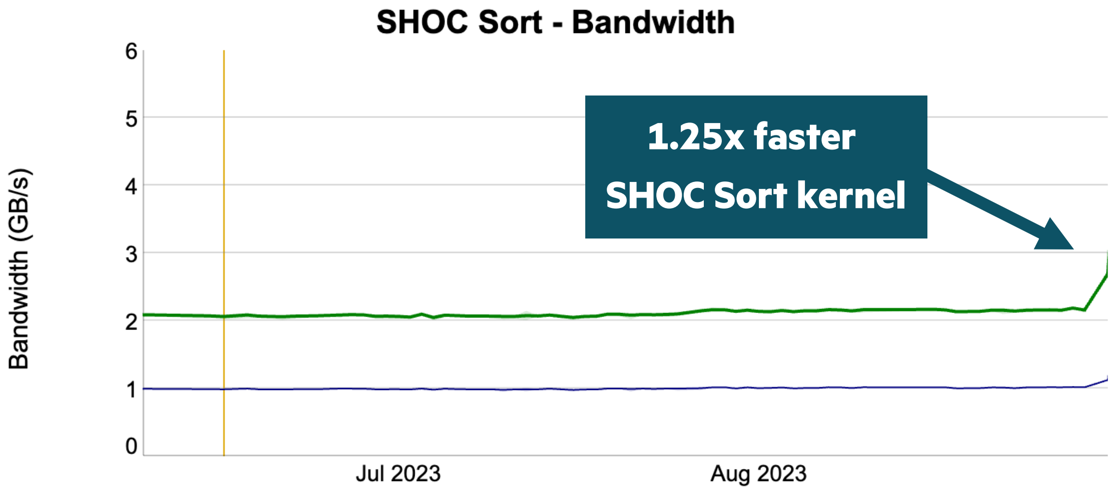
FASTER ARRAY ACCESS IN KERNELS

Impact

- Performance improvements across multiple benchmarks

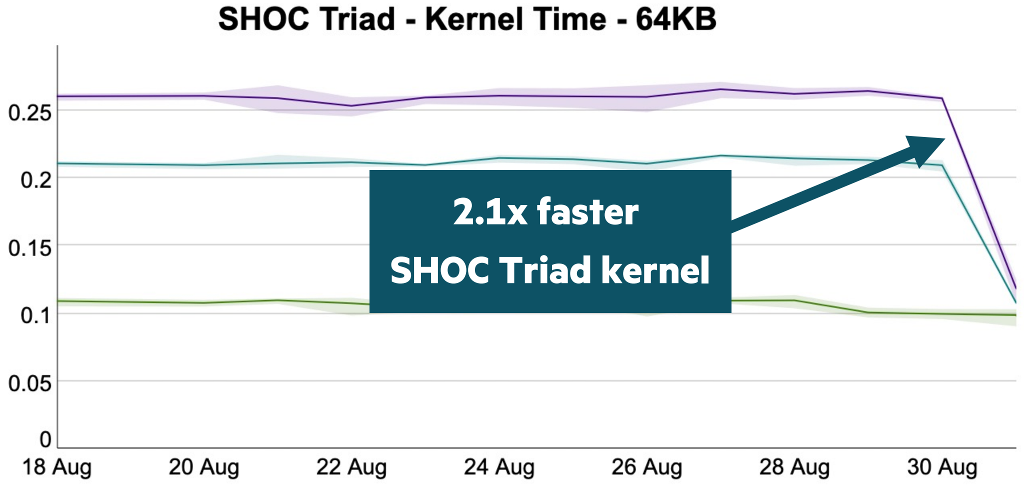
```
proc copyArray(ref A: [?D] int, ref B: [D] int) {
  foreach i in A.domain do B[i] = A[i];
}
```

4 metadata accesses *total!*



SHOC Sort - Bandwidth

1.25x faster SHOC Sort kernel



SHOC Triad - Kernel Time - 64KB

2.1x faster SHOC Triad kernel

PEER-TO-PEER ACCESS

Background and This Effort

Background:

- GPUs can communicate directly with one other
  - Can be through PCIe or communication links such as NVLink or Infinity Fabric
- Previously, Chapel's GPU runtime would not enable peer-to-peer communication

This Effort: Create a way to enable peer-to-peer communication

- Added the 'enableGpuP2P' config constant to 'GPU' module
  - To use, run your Chapel program with '--enableGpuP2P=true'

PEER-TO-PEER ACCESS

Impact

Impact: On NVIDIA, we see close to 6x throughput improvement in GPU-to-GPU transfers

- Tables measure 8 GiB transfers on a system with 4 NVIDIA A100-SXM4 GPUs
- Row and column correspond to source and destination GPU
- Each transfer was performed individually

Throughput (GiB/s)

enableGpuP2P=false

	0	**1**	**2**	**3**
0		13.2	11.9	13.3
1	13.2		13.4	13.5
2	13.2	13.3		13.5
3	13.2	13.2	13.4	

Throughput (GiB/s)

enableGpuP2P=true

	0	**1**	**2**	**3**
0		86.2	86.3	86.3
1	86.1		86.4	86.3
2	86.6	86.5		86.1
3	86.6	86.5	86.5	

PEER-TO-PEER ACCESS

Status and Next Steps

Status: While NVIDIA GPUs benefit from '--enableGpuP2P', AMD GPUs do not

- We have observed that AMD conducts peer-to-peer transfers by default
  - On Frontier we see ~10–47 GiB/s transfers in our benchmark regardless of how '--enableGpuP2P' is set
- With AMD, setting 'HSA_ENABLE_SDMA=0' adjusts GPU-to-GPU transfers for higher throughput
  - We observed up to 160 GiB/s transfer rates on Frontier with this setting

Next Steps:

- Find non-artificial benchmarks using peer-to-peer communication
- Further investigate peer-to-peer performance with AMD GPUs and Infinity Fabric
  - Determine if we want Chapel to adjust 'HSA_ENABLE_SDMA'
- Determine if we should allow turning on/off peer-to-peer access on an individual GPU level (#23621)
  - Or allow specifying peer-to-peer communication on an individual put/get basis

ARRAY-ON-DEVICE

Background:

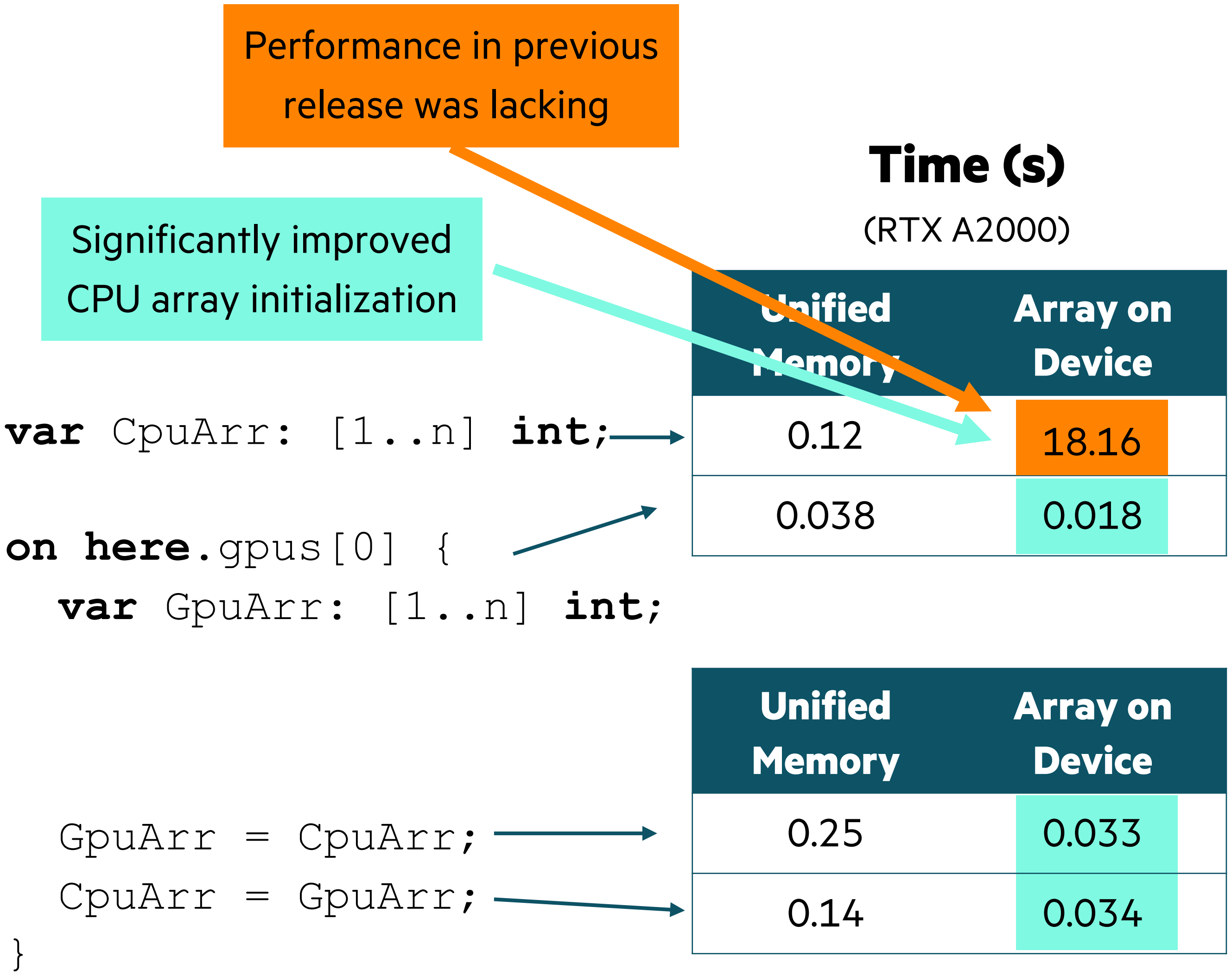
- 'array_on_device' is a memory strategy
  - Faster data transfers and GPU array initialization
  - However, CPU array initialization was sub-optimal

This Effort:

- Significantly improved performance
  - Implemented GPU-aware GET/PUT calls
  - This will also help GPU-driven communication

Status:

- 'array_on_device' performs better
  - 1.2x – 14x improvements in nightly testing
- It is the default memory strategy as of 1.32



Performance in previous release was lacking

Significantly improved CPU array initialization

Time (s)

(RTX A2000)

```
var CpuArr: [1..n] int;

on here.gpus[0] {
  var GpuArr: [1..n] int;

  GpuArr = CpuArr;
  CpuArr = GpuArr;
}
```

Unified Memory	Array on Device
0.12	18.16
0.038	0.018

Unified Memory	Array on Device
0.25	0.033
0.14	0.034

AVOIDING TASK STARVATION

Background

- Communication and computation overlap is:
  - An optimization to make use of different HW units
  - An important technique in GPU programming
- Chapel tasks are a natural way to achieve overlap
  - However, before 1.32 task starvation prevented that

This copy in 'begin' must wait b/c:

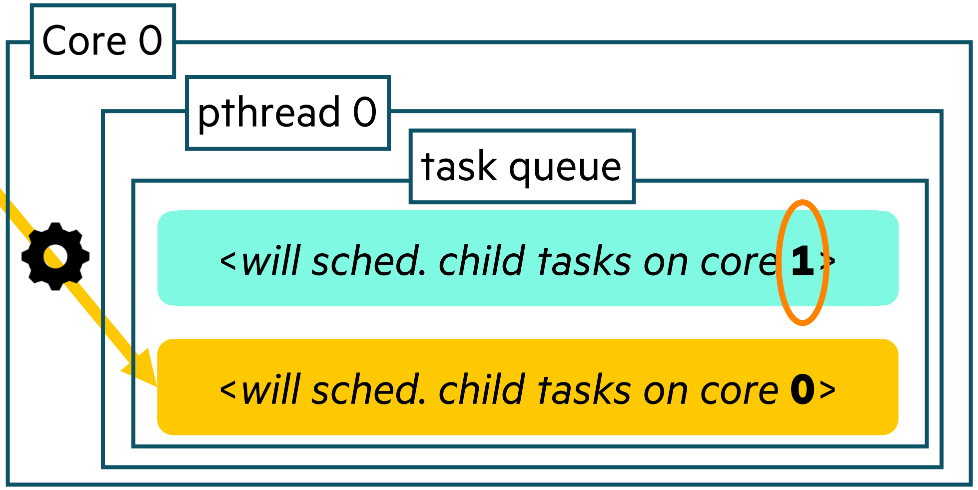
- it got scheduled behind the parent task
- current scheduler does not allow task stealing

... which will happen only when it hits sync variable read

Result: *No Overlap*

Task-private counter determines
the core on which child tasks will be scheduled

```
on here.gpus[0] {
  begin {
    gpuData2 = cpuData2;
    data2Copied.writeEF(true);   }
  foreach d in gpuData1 do foo(d);

  if data2Copied.readFE() then
    foreach d in gpuData2 do bar(d);
}
```



AVOIDING TASK STARVATION

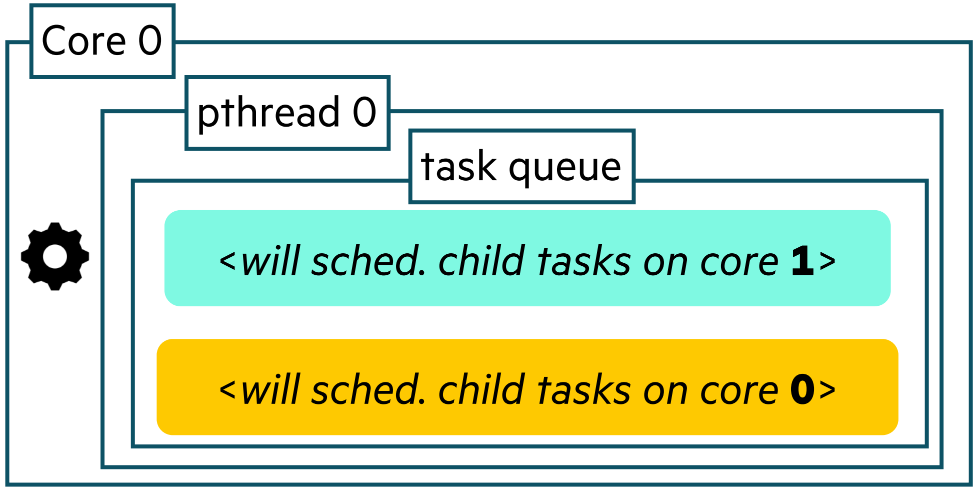
This Effort

- With 1.32, a task yields right after launching a kernel
  - In non-contentious cases, cost is not observable
  - If tasks contend for a core, allows overlap

```
on here.gpus[0] {
  begin {
    gpuData2 = cpuData2;
    data2Copied.writeEF(true);   }
  foreach d in gpuData1 do foo(d);

  if data2Copied.readFE() then
    foreach d in gpuData2 do bar(d);
}
```

The parent task will yield after launching the kernel

Result: *Overlap!*



The copy in 'begin' can execute

The main task waits in the queue while GPU kernel is executing

TASK-PARALLEL GPU OPERATIONS

Background

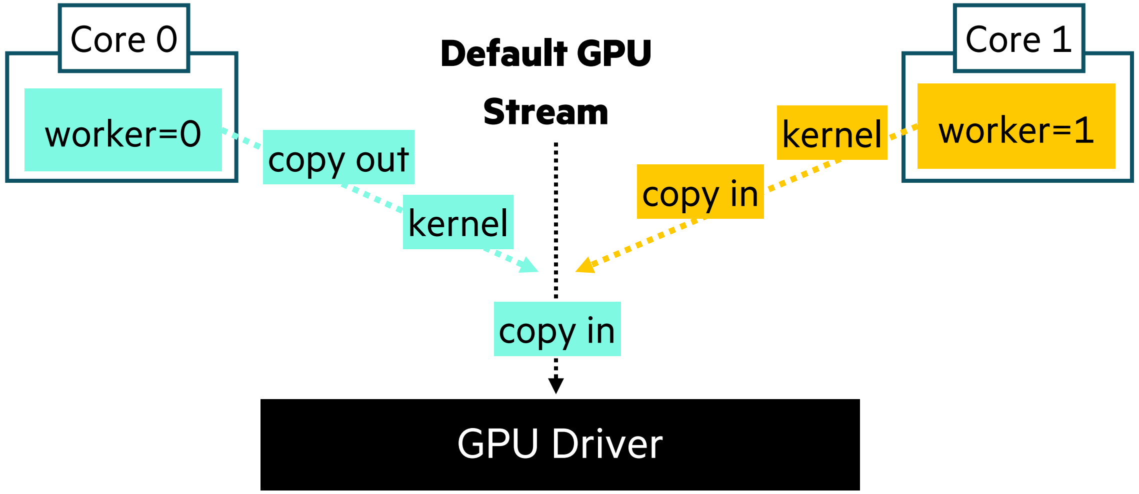
- To overlap communication/computation on a GPU:
  - Data can be split into chunks
  - Multiple *CUDA/HIP streams* do copy+kernel launch
  - GPU driver can interleave copies with launches
    - But they must come from different GPU streams
- One way of doing that in Chapel is:
  - Create multiple worker tasks per GPU
  - Have each of them run a loop
  - While picking the next chunk dynamically
  - Until all the chunks are processed
- Before 1.32, this would perform worse
  - Non-overlapped version is faster
  - Regardless of per-task size and/or number of tasks

```
on here.gpus[0] {
 coforall worker in 0..#numWorkers {
  var DevIn, DevOut: [0..#tSize] real;

  while true {
   // dynamically pick the next chunk
   const myChunkId = curChunk.fetchAdd(1);
   if myChunkId >= numChunks then break;

   const myChunk = myChunkId*tSize..#tSize;

   DevIn = HostIn[myChunk];      // copy in
   kernel(DevIn, DevOut);        // kernel
   HostOut[myChunk] = DevOut;    // copy out
  }
 }
}
```

TASK-PARALLEL GPU OPERATIONS

Background

- Previously, Chapel used the default GPU stream
  - i.e., GPU operations from parallel tasks got serialized



Core 0

worker=0

copy out

kernel

Default GPU Stream

copy in

kernel

copy in

Core 1

worker=1

GPU Driver

Results in completely sequential order:

time →

ops	t0	t1	t2	t3	t4	t5
xfer	in		in		out	out
exec		kernel		kernel		

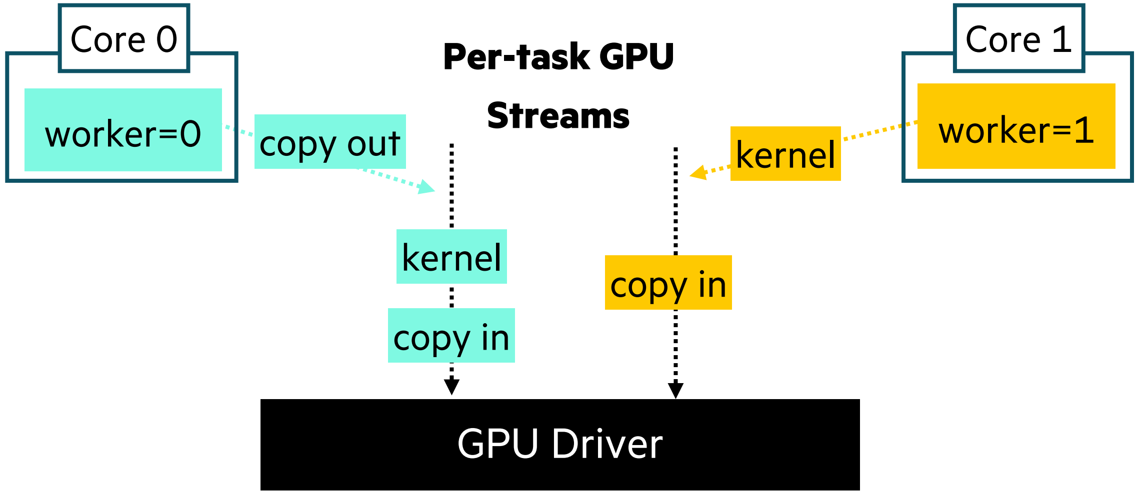
```
on here.gpus[0] {
 coforall worker in 0..#numWorkers {
  var DevIn, DevOut: [0..#tSize] real;

  while true {
   // dynamically pick the next chunk
   const myChunkId = curChunk.fetchAdd(1);
   if myChunkId >= numChunks then break;

   const myChunk = myChunkId*tSize..#tSize;

   DevIn = HostIn[myChunk];    // copy in
   kernel(DevIn, DevOut);      // kernel
   HostOut[myChunk] = DevOut;  // copy out
  }
 }
}
```

TASK-PARALLEL GPU OPERATIONS

This Effort and Impact

This Effort: Per-task, per-device streams

- Each worker task will have its own GPU stream



Enables better overlap:

time →

ops	t0	t1	t2	t3	t4	t5
xfer	in	in	out	out		
exec		kernel	kernel			

```
on here.gpus[0] {
 coforall worker in 0..#numWorkers {
  var DevIn, DevOut: [0..#tSize] real;

  while true {
   // dynamically pick the next chunk
   const myChunkId = curChunk.fetchAdd(1);
   if myChunkId >= numChunks then break;

   const myChunk = myChunkId*tSize..#tSize;

   DevIn = HostIn[myChunk];      // copy in
   kernel(DevIn, DevOut);        // kernel
   HostOut[myChunk] = DevOut;    // copy out
  }
 }
}
```

FASTER MATH LIBRARY CALLS IN KERNELS

Background: Math library calls like 'sqrt' were unexpectedly slower compared to CUDA/HIP

- Reported by a user (#22112)

This Effort: The performance issue is fixed in 1.32

- The compiler was generating calls that were wrapped in some helper functions that should have been inlined
- The root issue was the ordering of device library linkage w.r.t. the LLVM optimization pipeline

Impact: Math library functions perform on-par with CUDA/HIP

- Two mini-applications benefitted from this optimization

	Speedup	
	NVIDIA A100	**AMD MI250X**
coral	1.80x	1.25x
miniBUDE*	1.82x	1.92x

* https://github.com/xianghao-wang/miniBUDE/tree/benchmark

GPU SPECIALIZATION

Background

- GPU-eligible loops exhibit different behavior depending on if you are on a GPU locale or not
  - Namely, if we are on a GPU locale then we do a kernel launch
- Checking to see if we are on a GPU adds overhead at every eligible loop
  - Note the repeated execution of the 'if' statement in this example:

Original code:

```
on loc do
  for i in 0..<N do
    foreach k in 0..<P do ...
```

body of 'on' statement is outlined into a function

since the 'foreach' loop is GPU-eligible, we insert a runtime check to see if we are on a GPU locale. If so, launch it as a kernel.

The compiler "lowers" this to:

```
on loc do onBody()

proc onBody() do

  for i in 0..<N do

    if is_gpu_locale(here) then
      gpu_kernel_launch(extractedLoopFunc, ...)
    else
      foreach k in 0..<P do ...
```

GPU SPECIALIZATION

This Effort

- Clone functions reachable from 'on' statements into ”GPU-specialized" and "non-GPU-specialized" copies
  - Rewrite calls in GPU-specialized functions to call other specialized functions
  - Perform a runtime check to see if you are on a GPU in the 'on' statement; if so, call the cloned function

```
on loc do
  for i in 0..<N do
    foreach k in 0..<P do ...
```

Lowers to

```
if is_gpu_locale(loc) on loc do onGpuBody();
else                  on loc do onBody();
```

this specialization can avoid the runtime check

```
proc onGpuBody() do
 for i in 0..<N do
   gpu_kernel_launch(extractedLoopFunc, ...)
```

in theory this could too, but we don't currently account for virtual function calls, so we still do it here

```
proc onBody() do
  for i in 0..<N do
    if current_sub_locale >= 0 then
      gpu_kernel_launch(extractedLoopFunc, ...)
    else
      foreach k in 0..<P do ...
```

GPU SPECIALIZATION

Impact and Status

Impact: Current limitations prevent us from improving performance

- In an unsafe version, we see a 3x performance improvement
  - Unsafe because it does not rewrite virtual function calls in GPU-specialized functions to call GPU-specialized clones
- In our current safe version of the transform, we do not see a performance improvement
  - Safe because we do not remove 'if' statements from non-GPU-specialized functions
- Adding extra functions also increases compile time (~30% longer in some cases)

Status: The transform is considered experimental and may be beneficial in the future

- It can optionally be turned on by passing '--gpu-specialization' to 'chpl'
- Aside from removing a per-eligible-loop runtime check, the transform may prove useful for other optimizations:
  - specializing reductions on GPU locales
  - less aggressive wide pointer usage

GPU SPECIALIZATION

Next Steps

- Study more benchmarks, examining overhead from using the GPU locale model on non-GPU bound code
- Make the transform cognizant of virtual function calls
- Avoid overspecialization when unnecessary
- Explore other kinds of specialization that may not add as much compile-time overhead

SUMMARY & NEXT STEPS

GPU SUPPORT

Summary: Highlights from 1.31 and 1.32

Performance:

- Faster default memory strategy: 1.2x – 14x improvement on several benchmarks
- Faster array access in kernels: 1.1x – 2x improvement on several benchmarks
- Faster Math library calls: 1.3x – 1.9x improvement on two applications
- Can reach peak peer-to-peer bandwidth on Frontier

Portability:

- Feature and performance parity between NVIDIA and AMD targets
- CPU-as-Device mode

Features:

- Atomic operations
- Ability to compile for multiple NVIDIA architectures
- Increased introspection through: '--report-gpu', '--savec' on AMD and improved kernel naming

GPU SUPPORT

Proposed Next Steps for 1.33 and 1.34

Features:

- Foreach intents and better shadowing
- Warp-/wavefront-level functions
  - warp-synchronization
  - data shuffle
- Initial support for basic whole-array reductions
- Prototype syntax for advanced forall features

Performance:

- Continue investigating low-performance cases
- Investigate non-GPU execution performance
- Outer-loop vectorization for CPU

Portability:

- Improve cpu-as-device behavior parity
- Improve CUDA 12/ROCm 5 support with LLVM 16

Explorations:

- Try using dpc++ as the system LLVM for Intel GPUs
- Start working on GPU-driven communication
- Investigate launching multidimensional grids
- Start improving CPU/GPU portability

OTHER GPU IMPROVEMENTS

OTHER GPU IMPROVEMENTS

For a more complete list of GPU support changes and improvements in the 1.31 and 1.32 releases, refer to the following sections in the CHANGES.md file:

- ‘GPU Computing’
- ‘Bug Fixes for GPU Computing’

THANK YOU

https://chapel-lang.org
@ChapelLanguage