Hewlett Packard Enterprise

CHAPEL 1.27.0/1.28.0 RELEASE NOTES: ONGOING EFFORTS—GPU SUPPORT

Chapel Team

June 30, 2022 / September 15, 2022

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
  - Chapel is not yet able to directly use the GPUs on a system like Frontier, but that's our goal
    - today, such GPUs are only accessible in Chapel via its interoperability features

- In recent releases, we've…

  *…moved from an idea (**1.23**), …*

  *…to a demo (**1.24**), …*

  *…to a user-accessible feature (**1.25**), …*

  *…to being able to drive multiple GPUs on one locale (**1.26**).*

  - ***1.27**: Adds support across multiple locales, and improves diagnostics*
  - ***1.28**: Includes exploratory work on vendor portability (AMD), memory management, and benchmarking*

GPU SUPPORT

This Effort: Overview of Changes in 1.27 and 1.28

New Features and Capabilities:

- Multi-locale support
- Expanded loop eligibility
- Diagnostics and utility modules
- Internal-facing work (primitives and pragmas)
- Support for LLVM 14

Bug Fixes:

- Fixed a bug preventing the use of CUDA 10.1
- Fixed a bug preventing associative domain iteration
- No more "unresolved extern" warning
- No more "unknown CUDA version" warning
- Fixed bugs for 'locale.name'/'.numPUs' returning bad values on parent locales

Explorations:

- Vendor portability, specifically for AMD GPUs
- Memory strategies
- SHOC benchmarks (Triad and Sort)
- Performance tracking infrastructure

Outreach:

- Collaborations with Arkouda and ORNL
- Talk at CHIUW 2022

GPU SUPPORT

- New Features and Capabilities
  - Multi-locale Support
  - Newly GPU Eligible Loops
  - Diagnostics and Utilities
- Explorations
- Status Summary and Proposed Priorities

NEW FEATURES AND CAPABILITIES:
MULTI-LOCALE SUPPORT

MULTI-LOCALE SUPPORT
Background, Effort, and Impact

Background:

- Early efforts only supported the first GPU on the first node
- In 1.26 we added multi-GPU support on the first node
  - but still required 'CHPL_COMM=none'

This Effort: Added support for 'gasnet' and 'ibv' communication layers

Impact: Now possible to write native Chapel code that runs across all GPUs on a multi-node system

```
coforall loc in Locales do on loc {
  coforall gpu in here.gpus do on gpu {
    forall {
      // body of loop turns into GPU kernel
    }
  }
}
```

MULTI-LOCALE SUPPORT

Example

```
var A, B, C: [1..n] int;    // local arrays stored on locale 0
coforall node in Locales do on node {
  const locChunk = …;
  var Al: [locChunk] int;
  var Bl = B[locChunk], Cl = C[locChunk];

  coforall gpu in here.gpus do on gpu {
    const gpuChunk = …;
    var Ag: [gpuChunk] int;
    var Bg = Bl[gpuChunk], Cg = Cl[gpuChunk];

    Ag = Bg + alpha * Cg;

    Al[gpuChunk] = Ag;
  }
  A[locChunk] = Al;
}
```

Loops like on the previous slide

Perform computation (promotion turns into GPU kernel)

MULTI-LOCALE SUPPORT
Example

```
var A, B, C: [1..n] int;    // local arrays stored on locale 0
coforall node in Locales do on node {
  const locChunk = …;
  var Al: [locChunk] int;
  var Bl = B[locChunk], Cl = C[locChunk];

  coforall gpu in here.gpus do on gpu {
    const gpuChunk = …;
    var Ag: [gpuChunk] int;
    var Bg = Bl[gpuChunk], Cg = Cl[gpuChunk];

    Ag = Bg + alpha * Cg;

    Al[gpuChunk] = Ag;
  }
  A[locChunk] = Al;
}
```

'Al', 'Bl', and 'Cl' are node-local copied "chunks" of A,B, and C

'Ag', 'Bg', and 'Cg' and GPU-local copied "chunks" of Al, Bl, and Cl

NEW FEATURES AND CAPABILITIES: NEWLY GPU-ELIGIBLE LOOPS

NEWLY GPU-ELIGIBLE LOOPS
Background and This Effort

Background:

- 'chpl' compiler conducts an analysis to determine when a loop is eligible to become a GPU kernel
  - Non-eligible loops will execute on the CPU instead
- Known limitations are documented in the GPU tech note
- We plan to address many of these limitations in future releases

This Effort:

- Addressed loop eligibility limitations encountered while porting the SHOC benchmarks to Chapel
  - Several minor usability improvements (shown on next slide)
  - 'forall' over multidimensional arrays

NEWLY GPU-ELIGIBLE LOOPS

Impact

Impact:

- This loop is now eligible for GPU execution
- Comments indicate what now works

```
var A: [0..N] real;
var cond = funcReturningABool();
forall i in 0..10 {
  var tup  = (1,2);
  var rec  = someRecord();
  A[i] = A[i] * sin(pi);   // math functions
  if cond {                // certain types of 'if' statements
    // …
  }
  A[i] = A[i] + rec.prop;     // field accesses
  A[i+1] = A[i+1] + tup[1];   // use of tuples
}
```

FORALL OVER MULTIDIMENSIONAL ARRAYS

Background and This Effort

Background:

- Prior to 1.28, compiling GPU-bound loops over multidimensional arrays resulted in a compiler error

```
on here.gpus[0] {
  var A: [1..100, 1..100] int;
  forall a in A {
    a += 1
  }
}
```

This effort:

- In 1.28, the code works:
  - The iteration over the first dimension in the domain will be launched on the GPU
  - The iteration over the remaining dimension(s) is performed serially, as if it were a regular 'for' loop

NEW FEATURES AND CAPABILITIES: DIAGNOSTICS AND UTILITIES

DIAGNOSTICS AND UTILITIES
Background and This Effort

Background:

- Logging and assertion functions are useful to:
  - understand program behavior
  - get assurance that things run as you expect
  - help optimize for performance
- GPU support is an area that can definitely benefit from such tools

This Effort:

- Introduces a new module to track kernel launches: '**GPUDiagnostics**'
- '**Memory.Diagnostics**' now tracks allocations on GPUs
- Adds additional utilities in a new module: '**GPU**'
  - one notable feature is 'assertOnGpu()', which is used to ensure a loop executes on a GPU
- More details in the GPU tech note

DIAGNOSTICS AND UTILITIES

GPUDiagnostics module: start/stop verbose output

```
use GPUDiagnostics;

startVerboseGPU();          // start reporting GPU events (kernel launches)
on here.gpus[0] {
  var A: [0..10] int;
  foreach a in A do a += 1;    // this will launch as a kernel
}
stopVerboseGPU();           // stop reporting GPU events (kernel launches)
```

Output:

```
0 (gpu 0): foo.chpl:6: kernel launch (block size: 512x1x1)
```

DIAGNOSTICS AND UTILITIES

GPUDiagnostics module: counting kernel launches

```
use GPUDiagnostics;

startGPUDiagnostics();    // start counting GPU events (kernel launches)
on here.gpus[0] {
  var A: [0..10] int;
  foreach a in A do a += 1;    // this will launch as a kernel
}
stopGPUDiagnostics();     // stop counting GPU events (kernel launches)
writeln(getGPUDiagnostics());
```

Output:

```
(kernel_launch = 1)
```

DIAGNOSTICS AND UTILITIES

Memory.Diagnostics: new support for GPUs

```
use Memory.Diagnostics;
startVerboseMem();    // start reporting memory events
on here.gpus[0] {
  var A: [0..10] int;
  foreach a in A do a += 1;
}
stopVerboseMem();     // stop reporting memory events
```

Output:

```
0 (gpu 0): foo.chpl:4: allocate 88B of domain(1,int(64),false) at 0x7f90e8000800
0 (gpu 0): foo.chpl:4: allocate 168B of [domain(1,int(64),false)] int(64) at 0x7f90e8000a00
0 (gpu 0): foo.chpl:4: allocate 88B of array elements at 0x7f90e8000c00
0 (gpu 0): foo.chpl:5: free 88B of array elements at 0x7f90e8000c00
0 (gpu 0): foo.chpl:5: free 168B of [domain(1,int(64),false)] int(64) at 0x7f90e8000a00
0 (gpu 0): foo.chpl:5: free 88B of domain(1,int(64),false) at 0x7f90e8000800
```

DIAGNOSTICS AND UTILITIES

assertOnGpu()

Example asserting at compile-time:

```
proc directlyRecursiveFunc() { directlyRecursiveFunc(); }
foreach i in 0..10 {
  assertOnGpu();
  directlyRecursiveFunc();
}
// error: Loop containing assertOnGpu( ) is not eligible for execution on a GPU
// assertOnFailToGpuize.chpl:1: note: function is recursive
```

Example asserting at runtime:

```
on functionThatReturnsSomeLocale() {
  foreach i in 0..10 {
    assertOnGpu();
    // …
  }
}
// will halt at the assertion at runtime if 'functionThatReturnsSomeLocale( )' does not return a GPU locale
```

EXPLORATIONS

- GPU Vendor Portability
- Benchmarks and Performance Tracking
- PGAS Style Communication and GPUs
- Memory Strategies

EXPLORATIONS:
GPU VENDOR PORTABILITY

EXPLORATIONS
GPU Vendor Portability

Background:

- We currently only support NVIDIA GPUs, but want to support other vendors as well (e.g., AMD and Intel)

This Effort:

- Investigated a few options to achieve vendor portability
  - A) Write different runtime layers for each vendor
  - B) Use a portable library (e.g., 'libomptarget') as a portable runtime layer

Status:

- After investigating both options, we have decided to start with option A
- Removed vendor-specific code from main GPU API, pushing it into a smaller vendor-specific interface

Next Steps:

- Implement the vendor-specific interface for AMD and bring it up to par with NVIDIA
- Begin benchmarking the AMD layer and continue to optimize both

EXPLORATIONS:
BENCHMARKING AND PERFORMANCE TRACKING

BENCHMARKS AND PERFORMANCE TRACKING

Background and Effort

Background on benchmarking: We want benchmarks that target GPUs

- Ideally with base versions created and maintained by someone outside of our group
- Why we want benchmarks:
  - performance comparison
  - evaluate language expressibility
  - help guide our design
  - more robust test suite

Background on performance:

- With large datasets, we are close to matching the performance of a CUDA-based implementation of Stream
  - Stream is a benchmark that operates on vectors and scalars ('A = B + alpha * C')
- We want to evaluate (and maintain) our performance across different patterns

This Effort:

- Created Chapel version of SHOC Triad and Sort benchmarks
- Set up performance tracking infrastructure for GPUs

BENCHMARKS AND PERFORMANCE TRACKING

SHOC Benchmarks

- **SHOC:** The Scalable HeterOgeneous Computing Benchmark Suite
  - Developed by ORNL
  - Used to test performance and stability of GPUs

- Implemented single-GPU version of these two benchmarks:
  - Triad
    - uses a pipelining (computation/communication overlap) pattern not seen in our existing GPU implementations of Stream
    - we implemented both a "direct translation" version and a Chapeltastic version
  - Sort
    - radix sort
    - implemented a "direct translation" version; making a Chapeltastic version is future work

BENCHMARKS AND PERFORMANCE TRACKING

Impact and Next Steps

Impact:

- While implementing Sort we encountered bugs and ran into limitations
  - for example: allowing different block sizes on different kernels (this could only be configured on a whole-program basis)
- We created workarounds in the interim, which will eventually be exposed through the language
- We have also started gathering nightly performance data

Next steps:

- Continue implementation of SHOC benchmarks
- Implement benchmarks in other suites (e.g., RajaPerf)
- Create versions of benchmarks that target multiple nodes and GPUs
- Performance analysis and optimization

EXPLORATIONS:
PGAS-STYLE COMMUNICATION AND GPUS

PGAS-STYLE COMMUNICATION AND GPUS

Background

Background: Chapel's global namespace allows direct access to local and remote variables

- Having a global namespace simplifies parallel programming
- This means (outside of GPUs):
  - across nodes: no need to write MPI-style explicit send/receive calls to manage data migration
- The dream (for GPUs):
  - between GPUs and hosts: No need to write 'cuMemCpyHtoD' and the like
  - between GPUs: No need to write combinations of these things
- Communication layers such as GASNet are middleware layers that enable this outside of GPUs
  - Can we use them for GPUs?

PGAS-STYLE COMMUNICATION AND GPUS

This Effort and Next Steps

This Effort: investigating whether we can leverage GASNet; also identify new communication patterns

- GASNet does have support for accessing data on GPUs (i.e., support for memory kinds)
- However, it cannot address calls originating from within a GPU kernel

Next Steps:

- Potential solutions:
  - Have GPU signal back to CPU to conduct communication

- Other approaches:
  - Prefetch communication (hoist relevant writes/reads out of kernel)
  - Stop kernel, conduct communication, launch a new kernel to resume

EXPLORATIONS:
MEMORY STRATEGIES

MEMORY STRATEGIES
Background and This Effort

Background:

- By default, we use unified memory (a.k.a. "managed memory" or "Unified Virtual Memory")
  - we did this to implement GPU support quickly
  - in this mode, the CUDA driver migrates pages between physical host/device memories
- But there is a cost:
  - the compiler and user have less control over data management (which may be required for good performance)
  - it’s not compatible with GASNet's memory-kinds support

This Effort:

- Introduced *memory strategies*, selected via a new ‘CHPL_GPU_MEM_STRATEGY’ environment variable
  - Traditional approach is named ‘unified_memory’ and remains the default
  - New ‘array_on_device’ mode causes:
    - array data to be stored on device
    - all other data to be stored on “page locked” host memory, permitting it to be accessed directly by the GPU

MEMORY STRATEGIES

Examples

- In both examples, the code is the same, but where we allocate—and when we transfer data—differs
  - expressions in purple indicate data on host, orange on device
- With the *unified memory* mode ('A' moves twice, 'x' moves once) —

```
on here.gpus[0] {
  var x = 123;                         // x allocated into unified memory (starting on host)
  var A: [0..10] int;                  // array data allocated into unified memory (starting on host)
  foreach i in 0..10 do A[i] = A[i] + x;  // computation on device; a page faults occur: 'A' and 'x' move to device
  writeln(A);                          // page fault occurs: A is transferred to host
}
```

- With the *array on device* mode ('A' copied once, 'x' accessed via DMA once) —

```
on here.gpus[0] {
  var x = 123;                         // x allocated onto host (since it's a scalar and not an array)
  var A: [0..10] int;                  // array data allocated on device (in page-locked fashion)
  foreach i in 0..10 do A[i] = A[i] + x;  // computation on device; x accessed by DMA
  writeln(A);                          // A is transferred to host
}
```

MEMORY STRATEGIES
Next Steps

Next steps:

- Consider other modes to allocate all / more data on the host
- Identify memory access patterns that work for unified memory, yet not when the array data is on the device:
  - For example: element-wise access like 'A[idx] = …' is not working as of today
- Evaluate performance to better understand impact

STATUS SUMMARY & PROPOSED PRIORITIES

STATUS SUMMARY AND PROPOSED PRIORITIES

Summary:

New features:

- multi-locale support
- improved diagnostics
- improved loop eligibility

Explorations:

- vendor portability
- benchmarking
- memory strategies
- communication

Next Steps:

- AMD support
- Performance analysis and optimization of initial user GPU codes
- Port benchmarks to identify performance and feature gaps

OTHER GPU IMPROVEMENTS

OTHER LIBRARY IMPROVEMENTS

For a more complete list of GPU improvements in the 1.27.0 and 1.28.0 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