CHAPEL 1.29.0/1.30.0 RELEASE NOTES: GPU SUPPORT

Chapel Team
December 15, 2022 / March 23, 2023
OUTLINE

- Background
- New User-facing Features
- AMD Support
- Performance
- Next Steps
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).

- 1.29 and 1.30 have primarily focused on performance and portability
  - **performance**: significantly improved the time to launch and execute kernels
  - **portability**: added support for AMD GPUs
    - 1.29: could generate binaries for AMD GPUs using low-level features
    - 1.30: raised level of abstraction to target a single locale’s AMD GPUs using Chapel, similar to NVIDIA GPUs
  - also new features for users and capabilities for developers
Vector Increment Example: Basics

```plaintext
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

```haskell
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'

```rust
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 each local GPU

A slice of the data is copied between host and device
Vector Increment Example: Multiple GPUs on Multiple Locales

```plaintext
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
This Effort: Overview of Changes in 1.29 and 1.30

Performance:
• Much faster kernel launch
• Faster execution across many benchmarks

Portability:
• AMD GPUs can now be used in a single locale

Bug fixes:
• Fixed segmentation faults in “array on device” mode
• Fixed error handling while generating the GPU binary
• Fixed a bug that prevented using ‘nil’
• Worked around a thread synchronization bug in ‘clang’

New Features and Capabilities:
• Early support for NVIDIA profiler and debuggers
• New functions to...
  – create block-shared arrays
  – synchronize threads in the same block
  – set the block size of a kernel
  – measure time in a kernel
  – write to the console from a kernel

Studies and Explorations:
• Application-level improvements in ChOp
• Application-level improvements in SHOC Sort
• Implemented SHOC Transpose
• Coral image analysis
New Utility Functions: Optimization and Advanced Features

**Background:** Optimized GPU codes tend to require advanced features
  - e.g., synchronization, block-shared memory

**This Effort:** Added new user-facing procedures to the ‘GPU’ module:

```plaintext
foreach i in 1..n {
    setBlockSize(128); // set the block size to 128; the argument must be a ‘param’
    var sharedArr = createSharedArray(int, 10); // create a block-shared array of 10 ints; size must be a ‘param’
    // ...
    syncThreads(); // synchronize the threads within the block
    // ...
}
```

**Status:** New procedures are unstable, along with the ‘GPU’ module as a whole

**Next Steps:** Develop a more Chapeltastic way of writing more advanced GPU code
GPU SUPPORT
New Feature: Enabling Efficient use of Nsight Compute Profiler

Background:
- Debugging and profiling GPU kernels are typically more difficult than CPU applications
  - I/O support is typically poor, execution model is less intuitive, esoteric challenges
- NVIDIA has numerous profilers, where NSight Compute is used for profiling kernel performance
  - While using profilers for Chapel in general is not very straightforward, focusing on kernels is easier
- Out-of-the-box: NSight Compute was able to show line-by-line hardware counters when '-g' was used
  - However, '--fast -g' thwarted assembler optimizations → reduced kernel performance → less valuable profiling

This Effort:
- Added the '--gpu-ptxas-enforce-optimizations' flag to ensure that assembler optimizations are enabled

Impact:
- Significant help while trying to understand performance of compiler-generated kernels
  - Kernel performance is virtually unaffected
  - Profiler shows line-by-line information accurately
- Can compare performance behavior of a reference version against the Chapel version
GPU SUPPORT
New Utility Functions: Debugging and Introspection

**Background:** Needed a way to trace code and measure performance when writing benchmarks

**This Effort:** Add 'gpuWrite()', 'gpuClock()', and 'gpuClocksPerSec()' procedures

```chapel
foreach i in 0..<N {
    gpuWrite(c"Start\n");  // gpuWrite() called on GPU; takes a c_string; output flushed on kernel termination
    const start = gpuClock();
    A = bigComputation(B);
    const end = gpuClock();
    gpuWrite(c"Stop\n");
    t1[i] = start; t2[i] = end;
}
const div = gpuClocksPerSec(0);  // gpuClocksPerSec() must be called on the host; passed GPU device ID
writeln("Time took: ", (t2[0] - t1[0]):real / div:real));
```

**Next Steps:** Replace these routines with stabilized versions in a future release
- Remove 'gpuWrite()' and get 'write()'/'writeln()'/'writef()' working inside GPU kernels
- Remove 'gpuClock()' and get Chapel's 'stopwatch' working inside GPU kernels
Targeting AMD GPUs

**Background:** In previous releases we only supported NVIDIA GPUs
- However, we intend for Chapel’s GPU support to run on devices from diverse vendors

**This Effort:** Add support for AMD GPUs
- Added 'CHPL_GPU_CODEGEN' to choose between working with an AMD or NVIDIA GPU
  - it will be set automatically if 'nvcc' or 'hipcc' are present
- Note that AMD GPU support requires the 'AMDGPU' LLVM target and the ROCM/HIP runtime libraries
  - the 'AMDGPU' LLVM is included in the bundled LLVM
  - the ROCM/HIP runtime is packaged as part of 'hipcc'

**Impact:**
- Now you can use Chapel to write code that runs on AMD GPUs
- Chapel code that was working on an NVIDIA GPU can be run on an AMD GPU without changing the code
GPU SUPPORT
Targeting AMD GPUs

Status:

- AMD GPU support has feature parity with NVIDIA GPU support except for:
  - certain 64-bit math functions
  - multilocal support: the AMD GPUs currently only work on a single locale ('CHPL_COMM=none')
- Applications are compiled to either run on NVIDIA GPUs or AMD GPUs, not both at once
- Performed first run on Frontier using HPCC Stream
  - Very close performance to baseline version at >10 TB/s bandwidth per node

Next Steps:

- Our aim is for Chapel to be completely portable between NVIDIA, AMD, and Intel GPUs
  - for AMD: support missing math functions and add multilocale support
  - for Intel: start adding support
- Consider using the 'hipify' tool to produce part (or all) of our AMD vendor-specific runtime
- Consider supporting a single binary that can run across GPUs from different vendors
PERFORMANCE
**Background:** Previously, the runtime would load the GPU binary whenever a kernel was launched
- This was mostly an artifact from earlier stages of development

**This Effort & Impact:** The GPU binary is loaded at application startup time
- Led to more than 300x faster kernel launch performance
- Significant improvements in HPCC Stream Triad with small vector sizes (see next slide)
GPU SUPPORT
Benefits from Loop-Invariant Code Motion (LICM)

Background:
- LICM is a compiler optimization to avoid redundantly performing a computation in a loop
- Chapel moves the body of GPU-eligible loops into separate kernel functions
- Some instances where LICM could improve performance were missed because we convert loops into kernels

This Effort:
- Solution: reorder how we do things to do LICM first

Impact:
- Can introduce extra arguments to kernel functions
- Improved performance of Stream and ChOp

Next Steps:
- Find and mitigate remaining overhead(s) in Stream
- Improve LICM for better GPU performance
**GPU SUPPORT**

Sidebar: Stream Performance with AMD

- Baseline was made by taking a CUDA implementation of Stream and running it through 'hipify'
- With array-on-device mode we see worse performance on small data sizes; more competitive on larger
- In unified memory mode, performance suffers; we have not yet investigated why this is
GPU SUPPORT
Communication Overlap

**Background:**
- GPUs can communicate and compute at the same time, and making use of that may improve performance
- In array-on-device mode, assignment statements perform synchronous (blocking) communication

**This Effort:**
- Explored how overlapped communication can be expressed in Chapel when in array-on-device mode
- Specifically, we created two Chapel versions of the SHOC Triad benchmark:
  - version 1: uses 'begin' statements and synchronization variables
  - version 2: adds explicit asynchronous communication routines to Chapel and uses them in the benchmark

**Status:**
- New API for asynchronous communication is implemented in the 'GPU' module but is undocumented
- Our Chapel versions do not yet show a benefit from using an overlap
- We have open questions about why the CUDA version does show a benefit
**Next Steps:**

- Consider adding asynchronous PUT and GET functions in the 'Communication' module
  - these could be generalized for both CPU-to-CPU and CPU-to-GPU communication

- Consider whether new language features would make such patterns easier to express

- Better understand why the CUDA version of SHOC Triad sees a benefit from asynchronous communication
  - or, find a better benchmark to demonstrate the value of computation/communication overlap

- Consider if using CUDA/HIP streams for regular allocations and launches can improve overall performance
Background: We have two memory strategies controlled by 'CHPL_GPU_MEM_STRATEGY'

- 'unified_memory' is the default strategy, relies on managed memory
  - Allows both host and device to access memory, where the underlying layer migrates pages between device and host
  - Easy to program, not ideal for performance
  - Some GPU features can't be used with this mode
- 'array_on_device': Closer to conventional GPU programming
  - Array data is allocated on device memory, where metadata is still on managed memory for easy initialization
  - Probably the only way to support GPU-driven communication in an efficient way
  - Our implementation showed promising performance in some cases, but also had segfaults

This Effort: Made progress towards making 'array_on_device' the default strategy

- Segfaults are fixed
- Investigated its performance and discovered some issues
## GPU SUPPORT

*array_on_device' Performance

<table>
<thead>
<tr>
<th>Time (s) (RTX A2000)</th>
<th>Unified Memory</th>
<th>Array on Device</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.12</td>
<td>18.16</td>
</tr>
<tr>
<td></td>
<td>0.038</td>
<td>0.018</td>
</tr>
</tbody>
</table>

**Faster initialization on GPU**

<table>
<thead>
<tr>
<th>Unified Memory</th>
<th>Array on Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>0.033</td>
</tr>
<tr>
<td>0.14</td>
<td>0.034</td>
</tr>
</tbody>
</table>

- **Array initialization on CPU is the next focus**

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

on here.gpus[0] {
    var GpuArr: [1..n] int;
    GpuArr = CpuArr;
    CpuArr = GpuArr;
}
```

**How memory is allocated**

<table>
<thead>
<tr>
<th>Unified Memory</th>
<th>Array on Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>metadata</td>
<td>host</td>
</tr>
<tr>
<td>host</td>
<td>host</td>
</tr>
<tr>
<td>data</td>
<td>host (registered)</td>
</tr>
<tr>
<td>managed</td>
<td>managed</td>
</tr>
<tr>
<td>managed</td>
<td>device</td>
</tr>
</tbody>
</table>
**Background:**

- **Chapel-based Optimization**
  - a user application that's part of our nightly performance tracking
  - branch-and-bound algorithms for combinatorial optimizations

**This Effort:**

- Initially Chapel was off by 10x from the reference version
  - with an application-level performance bug fixed, we were 2x off
- With the new profiler support, we profiled the Chapel version
  - application-level optimizations → ~1.8x improvement in Chapel
  - back-ported same optimizations to interop version → ~1.2x improvement
- We are about 15–20% off on NVIDIA

**Next Steps:**

- Investigate the source(s) of the remaining overhead
- Understand AMD performance better (in general and for ChOp)

---

**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>

(1x AMD MI100)

<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>0.40</td>
<td>0.55</td>
<td>36%</td>
</tr>
<tr>
<td>16</td>
<td>1.14</td>
<td>2.18</td>
<td>91%</td>
</tr>
<tr>
<td>17</td>
<td>6.36</td>
<td>13.28</td>
<td>209%</td>
</tr>
<tr>
<td>18</td>
<td>47.04</td>
<td>115.51</td>
<td>246%</td>
</tr>
</tbody>
</table>

*Tiago Carneiro, Nouredine Melab, Jan Gmys, Guillaume Helbecque. et al. — INRIA Lille, France; Imec, Belgium; University of Mons, Belgium; et al.
GPU SUPPORT
Study: SHOC-Sort and SHOC-Transpose

**SHOC-Sort:**
- A radix-sort implementation on the GPU
- Initial port was about 6–7x off from the base version
  - Dynamically creating and destroying lists on the host was a big source of overhead
- Fixing that, our implementation is closer to the base version in terms of behavior
  - currently, still 2x off

**SHOC-Transpose:**
- Tiled matrix transposition using shared memory
- We’ve implemented:
  - a naïve version, i.e.,
    ```c
    foreach (i,j) in Dom do A[i,j] = B[j,i];
    ```
  - an optimized implementation using tiling within the 'foreach' loop
  - a low-level version that uses non-user-facing ways to launch kernels
- The low-level version is within percentages of reference, others are 4x off
  - Naïve is expected to perform poorly over tiled one, the optimized version requires more investigation
Next Steps: Performance

• Fix expensive CPU array initialization on 'array_on_device' mode
  • This is expected to be resolved via a more general effort to improve CPU performance of the GPU locale model

• Investigate specializing AST for GPU code paths
  • This would involve code cloning/versioning during compilation to specialize code being executed on the GPU
  • Today, only the loop body is specialized by virtue of creating a kernel from it
    – 'on' statements or other functions called from the GPU aren’t specialized by the Chapel compiler

• Investigate loop-invariant code motion (LICM) improvements
  • Moving GPU transformation after LICM improved performance in many cases
    – However, LICM can be more aggressive, as we see invariants in GPU kernels in some cases
    – It can also help if LICM can reduce redundancy in cases where an array is accessed multiple times in a kernel

• Continue working on the benchmarks where performance is behind reference
GPU SUPPORT
Summary: Highlights from 1.29 and 1.30

• AMD GPUs can be used in single-locale settings
  • Feature/correctness parity with NVIDIA except for missing support for some 64-bit math
  • Initial performance tests didn’t point to any major issue, though it is behind NVIDIA in some cases
  • First run on Frontier using HPCC Stream
    – Very close performance to base at >10 TB/s bandwidth per node

• Significant performance improvements
  • 300x faster kernel launch
  • Performance optimizations that led to improvements in HPCC Stream, SHOC Triad, SHOC Sort, and ChOp
  • Application-level improvements in ChOP and SHOC Sort

• Usability improvements
  • Several new functions
    – gpuClock(), gpuWriteln(), setBlockSize(), createSharedArray(), syncThreads()
  • Initial support for debugger and profilers
**GPU SUPPORT**

Proposed Next Steps for 1.31 and 1.32

**Performance:**
- Continue investigating low-performance cases
- Fix 'array_on_device' performance issues
  - make it the default memory strategy
- Improve non-GPU execution performance
- Investigate streams for better CPU/GPU overlap
- Gain experience with NVLink and ensure its utilization

**Portability:**
- Start working towards Intel GPU support
- Gain experience with EX

**Features:**
- Make progress on distributed array support
- Make progress on design of new features
  - querying task/thread/vector lane ids
  - block-synchronization
  - shared memory allocation
- Outer-loop vectorization for CPU

**Explorations:**
- Shadow variables in GPU kernels
- User applications
  - CHAMPS, Coral Image Analysis
OTHER GPU IMPROVEMENTS

For a more complete list of GPU support changes and improvements in the 1.29.0 and 1.30.0 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