GPUIterator: Bridging the Gap between Chapel and GPU platforms

Akihiro Hayashi (Rice), Sri Raj Paul (Georgia Tech), Vivek Sarkar (Georgia Tech)
GPUs are a common source of performance improvement in HPC

Source: https://www.top500.org/statistics/list/
GPU Programming in Chapel

- Chapel’s multi-resolution concept
  - Start with writing “forall” loops (on CPU, proof-of-concept)
    ```chapel
    forall i in 1..n {
        ...
    }
    ```
  - Apply automatic GPU code generators [1][2] when/where possible
  - Consider writing GPU kernels using CUDA/OpenCL or other accelerator language, and invoke them from Chapel (Focus of this paper)

Motivation:
Vector Copy (Original)

```
1  var A: [1..n] real(32);
2  var B: [1..n] real(32);

3  // Vector Copy
4  forall i in 1..n {
5      A(i) = B(i);
6  }
```
Motivation:
Vector Copy (GPU)

- Invoking CUDA/OpenCL code using the C interoperability feature

```c
1 extern proc GPUVC(A: [] real(32),
2     B: [] real(32),
3     lo: int, hi: int);
4
5 var A: [1..n] real(32);
6 var B: [1..n] real(32);

8 // Invoking CUDA/OpenCL program
9 GPUVC(A, B, 1, n);
```

```c
1 // separate C file
2 void GPUVC(float *A,
3     float *B,
4     int start,
5     int end) {
6     // CUDA/OpenCL Code
7 }
```
Motivation:
The code is not very portable

Potential “portability” problems

- How to switch back and forth between the the original version and the GPU version?
- How to support hybrid execution?
- How to support distributed arrays?

Research Question:
What is an appropriate and portable programming interface that bridges the "forall" and GPU versions?
Our Solution: GPUIterator

Contributions:
- Design and implementation of the GPUIterator
- Performance evaluation of different CPU+GPU execution strategies

// Original Version
forall i in 1..n {
    A(i) = B(i);
}

// GPU Version
GUIVC(A, B, 1, n);

// GPU Iterator (in-between)
var G = lambda (lo: int, hi: int, nElems: int) {
    GPUVC(A, B, lo, hi);
};

var CPUPercent = 50;
forall i in GPU(1..n, G, CPUPercent) {
    A(i) = B(i);
}
Chapel’s iterator

Chapel’s iterator allows us to control over the scheduling of the loops in a productive manner.

```chapel
1 // Iterator over fibonacci numbers
2 forall i in fib(10) {
3   A(i) = B(i);
4 }
```

<table>
<thead>
<tr>
<th>CPU1</th>
<th>CPU2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>34</td>
</tr>
</tbody>
</table>

[https://chapel-lang.org/docs/master/primers/parIters.html](https://chapel-lang.org/docs/master/primers/parIters.html)
The GPUIterate automates work distribution across CPUs+GPUs.

```plaintext
1. forall i in 1..n { 
2.   A(i) = B(i);
3. }
```

```
1. forall i in GPU(1..n, GPUWrapper, CPUPercent) { 
2.   A(i) = B(i);
3. }
```

CPU Portion

```
1 2 3 4
CPU Portion
```

GPU Portion

```
CPU Portion
```

CPU1  CPU2  .............  CPUm

CPU1  CPUm

GPU1  .............  GPUk

CPUPercent

```
CPU Portion
```

GPUPercent = 100 - CPUPercent

```
GPU Portion
```

CPU1  CPUm

GPU1  .............  GPUk
How to use the GPUIterator?

1 var GPUCallBack = lambda (lo: int, hi: int, nElems: int){
2   assert(hi-lo+1 == nElems);
3   GPUVC(A, B, lo, hi);
4 }
5 forall i in GPU(1..n, GPUCallBack, CPUPercent) {
6   A(i) = B(i);
7 }

This callback function is called after the GPUIterator has computed the subspace (lo/hi: lower/upper bound, n: # of elements)

GPU() internally divides the original iteration space for CPUs and GPUs

Tip: declaring CPUPercent as a command-line override (“config const”) helps us to explore different CPU+GPU executions
The GPUIterator supports Distributed Arrays

```plaintext
var D: domain(1) dmapped Block(boundingBox={1..n}) = {1..n};
var A: [D] real(32);
var B: [D] real(32);
var GPUCallBack = lambda (lo: int, hi: int, nElems: int) {
    GPUVC(A.localSlice(lo..hi), B.localSlice(lo..hi), 0, hi-lo, nElems);
};
forall i in GPU(D, GPUCallBack, CPUPercent) {
    A(i) = B(i);
}
```
The GPUIterator supports Zippered-forall

Implementation of the GPUIterator

- **Internal modules**
  - [https://github.com/ahayashi/chapel](https://github.com/ahayashi/chapel)
  - Created the GPU Locale model
    - `CHPL_LOCALE_MODEL=gpu`

- **External modules**
  - [https://github.com/ahayashi/chapel-gpu](https://github.com/ahayashi/chapel-gpu)
  - Fully implemented in Chapel
Implementation of the GPUIterator

```cpp
coforall subloc in 0..1 {
  if (subloc == 0) {
    const numTasks = here.getChild(0).maxTaskPar;
    coforall tid in 0..#numTasks {
      const myIters = computeChunk(...);
      for i in myIters do
        yield i;
    }
  } else if (subloc == 1) {
    GPUCallBack(...);
  }
}
```
Writing CUDA/OpenCL Code for the GPUIterator

GPU programs for the GPUIterator should include typical host and device operations.
Performance Evaluations

 Platforms

- Intel Xeon CPU (12 cores) + NVIDIA Tesla M2050 GPU
- IBM POWER8 CPU (24 cores) + NVIDIA Tesla K80 GPU
- Intel Core i7 CPU (6 cores) + Intel UHD Graphics 630/AMD Radeon Pro 560X
- Intel Core i5 CPU (4 cores) + NVIDIA TITAN Xp

 Chapel Compilers & Options

- Chapel Compiler 1.20.0-pre (as of March 27) with the --fast option

 GPU Compilers

- CUDA: NVCC 7.0.27(M2050), 8.0.61 (K80) with the -O3 option
- OpenCL: Apple LLVM 10.0.0 with the -O3 option
Performance Evaluations (Cont’d)

- **Tasking**
  - CUDA: CHPL_TASK=qthreads
  - OpenCL: CHPL_TASK=fifo

- **Applications** ([https://github.com/ahayashi/chapel-gpu](https://github.com/ahayashi/chapel-gpu))
  - Vector Copy
  - Stream
  - BlackScholes
  - Logistic Regression
  - Matrix Multiplication
How many lines are added/modified?

<table>
<thead>
<tr>
<th></th>
<th>LOC added/modified (Chapel)</th>
<th>CUDA LOC (for NVIDIA GPUs)</th>
<th>OpenCL LOC (for Intel/AMD GPUs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Copy</td>
<td>6</td>
<td>53</td>
<td>256</td>
</tr>
<tr>
<td>Stream</td>
<td>6</td>
<td>56</td>
<td>280</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>6</td>
<td>131</td>
<td>352</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>11</td>
<td>97</td>
<td>472</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>6</td>
<td>213</td>
<td>290</td>
</tr>
</tbody>
</table>

Source code changes are minimal

LOC for CUDA/OpenCL are out of focus
The iterator enables exploring different CPU+GPU strategies with very low overheads.

The GPU is up to 145x faster than the CPU, but is slower than the GPU due to data transfer costs in some cases.
How fast are GPUs?
(Single-node, Xeon + M2050)

- The iterator enables exploring different CPU+GPU strategies with very low overheads.
- The GPU is up to 126x faster than the CPU, but is slower than the GPU due to data transfer costs in some cases.
How fast are GPUs compared to Chapel’s BLAS module on CPUs? (Single-node, Core i5 + Titan Xp)

Motivation: to verify how fast the GPU variants are compared to a highly-tuned Chapel-CPU variant

Result: the GPU variants are mostly faster than OpenBLAS’s gemm (4 core CPUs)
When is hybrid execution beneficial? (Single node, Core i7+UHD)

With tightly-coupled GPUs, hybrid execution is more beneficial.
Multi-node performance numbers (Xeon + M2050)

- The original forall show good scalability
- The GPU variants give further performance improvements

![Graph showing speedup over original forall for BlackScholes (Higher is better).]
Conclusions & Future Work

- **Summary**
  - The GPUIterator provides an appropriate interface between Chapel and accelerator programs
    - Source code is available:
      - https://github.com/ahayashi/chapel-gpu
  - The use of GPUs can significantly improve the performance of Chapel programs

- **Future Work**
  - Support reduction
  - Further performance evaluations on multi-node CPU+GPU systems
  - Automatic selection of the best “CPUPercent”
Backup Slides
GPU is not always faster

CPU: IBM POWER8 @ 3.69GHz, GPU: NVIDIA Tesla K40m
The GPUIterator supports Distributed Arrays (Cont’d)

- No additional modifications for supporting multi-locales executions

<table>
<thead>
<tr>
<th>Locale 0</th>
<th>A(x)</th>
<th>A(y)</th>
<th>Locale 1</th>
<th>A(z)</th>
<th>A(w)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Portion</td>
<td>GPU Portion</td>
<td>CPU Portion</td>
<td>GPU Portion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A (Chapel)</td>
<td>1</td>
<td><strong>A.localeSlice(lo..hi)</strong></td>
<td></td>
<td><strong>A.localeSlice(lo..hi)</strong></td>
<td>n</td>
</tr>
</tbody>
</table>

Note: localeSlice is Chapel’s array API