Advanced Topics in Numerical Analysis: High Performance Computing

Intro to GPGPU

Georg Stadler, Dhairya Malhotra
Courant Institute, NYU

Spring 2019, Monday, 5:10–7:00PM, WWH #1302

April 8, 2019
Outline

Organization issues

Final projects

Computing on GPUs
Organization

Scheduling:
- Homework assignment #4 due next Monday

Topics today:
- Final project overview/discussion
- More GPGPU programming (several examples)
- Algorithms: image filtering (convolution), parallel scan, bitonic sort

Outlook for next week(s):
- Distributed memory programming (MPI)
Outline

Organization issues

Final projects

Computing on GPUs
Final projects

- **Final projects!** Pitch/discuss your final project to/with us. We’re available Tuesday (tomorrow) 5-6pm and Thursday 11-12:30 in WWH #1111 or over Slack.
- Would like to (more or less) finalize project groups and topics in the next week.
- Final projects are in teams of 1-3 people (2 preferred!)
- We posted suggestions for final projects. More ideas on the next slides. Also, take a look at the HPC projects we collected from the first homework assignment.
- **Final project presentations** (max 10min each) in the week May 20/21. You are also required to hand in a short paper with your results, as well as the git repo with the code.
Final projects

Final project examples (from example list):

- Parallel multigrid
- Image denoising
- Adaptive finite volumes
- Parallel k-means
- Fluid mechanics simulation
- Data partitioning using parallel octrees
Final projects

Final project examples (more examples):

- Parallelizing a DFT sub-calculation (Tkatchenko-Scheffler dispersion energies and forces)
- Parallelizing a neural network color transfer method for images
- Parallel all-pairs shortest paths via Floyd-Warshall
- Fast CUDA kernels for ResNet inference
- ... Take an existing serious code and speed it up/parallelize it
- ...
Outline

Organization issues

Final projects

Computing on GPUs
Review of Last Class

- **CUDA programming model**: GPU architecture, memory-hierarchy, thread-hierarchy.
- **Shared memory**: fast, low-latency, shared within thread-block, 48KB - 128KB (depending on compute capability)
  - avoid bank conflicts within a warp.
- **Synchronization**
  - `__syncthreads()`: all threads in a block
  - `__syncwarp()`: all threads in a warp
- **Reduction on GPUs**
Hiding Latency

- All operations have latency
- CPUs hide latency using out-of-order computation and branch prediction; reduce latency of memory accesses using caches.
- GPUs hide latency using parallelism:
  - execute warp-1 (threads 0-31)
  - when warp-1 stalls, start executing warp-2 (threads 32-63) and so on · · ·
Occupancy Calculator

Get resource usage for kernel functions (compiler flag: `-Xptxas -v`)

**Example:**

```
# nvcc -std=c++11 -Xcompiler "-fopenmp" -Xptxas -v reduction.cu
```

ptxas info: Compiling entry function

'`Z16reduction_kernelPdPKdl' for 'sm_30'

ptxas info: Function properties for _Z16reduction_kernelPdPKdl
0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads

ptxas info: **Used 28 registers, 8192 bytes smem, 344 bytes cmem[0]**

\[
\text{Occupancy} = \frac{\text{#-of-threads per SM}}{\text{max-#-of-threads per SM}}
\]

- Calculate occupancy for your code:
  - web version: [https://xmartlabs.github.io/cuda-calculator](https://xmartlabs.github.io/cuda-calculator)
  - **Improve occupancy to improve performance**
Device Management (Multiple GPUs)

- Get number of GPUs: `cudaGetDeviceCount(int *count)`
- Set the current GPU: `cudaSetDevice(int device)`
- Get current GPU: `cudaGetDevice(int *device)`
- Get GPU properties:
  `cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`
Streams

- execute multiple tasks in parallel; either on separate GPUs or on the same GPU.
- useful for executing several independent small tasks where each task does not have sufficient parallelism.

```c
// create streams
cudaStream_t stream1, stream2
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

// launch two kernels in parallel
kernel<<<1, 64, 0, stream1>>>(());
kernelll<<1, 128, 0, stream2>>>(());

// synchronize
cudaStreamSynchronize(stream1)
cudaStreamSynchronize(stream2)
```
Image Filtering

**Convolution:** read $k \times k$ block of the image multiply by filter weights and sum.

figure from: GPU Computing: Image Convolution - Jan Novák, Gábor Liktor, Carsten Dachsbacher
Image Filtering

Using shared memory as cache to minimize global memory reads

- read a $32 \times 32$ block of original image from main memory
- compute convolution in shared memory
- write back result sub-block (excluding halo)

figure from: GPU Computing: Image Convolution - Jan Novák, Gábor Liktor, Carsten Dachsbacher
Comparison based sorting algorithms: bubble sort $\mathcal{O}(N^2)$, sample sort $\mathcal{O}(N \log N)$, merge sort $\mathcal{O}(N \log N)$

Bitonic merge sort $\mathcal{O}(N \log^2 N)$
- great for small to medium problem sizes.
- sorting networks, simple deterministic algorithm bases on compare and swap.
- sequence of $\log N$ bitonic merge operations.
**Sorting**

**Comparison based sorting algorithms:** bubble sort $O(N^2)$, sample sort $O(N \log N)$, merge sort $O(N \log N)$

**Bitonic merge sort** $O(N \log^2 N)$
- great for small to medium problem sizes.
- sorting networks, simple deterministic algorithm bases on compare and swap.
- sequence of $\log N$ bitonic merge operations.
Comparison based sorting algorithms: bubble sort $O(N^2)$, sample sort $O(N \log N)$, merge sort $O(N \log N)$

Bitonic merge sort $O(N \log^2 N)$

- Great for small to medium problem sizes.
- Sorting networks, simple deterministic algorithm bases on compare and swap.
- Sequence of $\log N$ bitonic merge operations.
Comparison based sorting algorithms: bubble sort $\mathcal{O}(N^2)$, sample sort $\mathcal{O}(N \log N)$, merge sort $\mathcal{O}(N \log N)$

**Bitonic merge sort** $\mathcal{O}(N \log^2 N)$
- great for small to medium problem sizes.
- sorting networks, simple deterministic algorithm bases on compare and swap.
- sequence of $\log N$ bitonic merge operations.
Bitonic Sort

**Bitonic merge** $O(N \log N)$ cost for each merge operation

- divide-and-conquer algorithm on bitonic sequences.
- **Bitonic sequence**: a sequence that changes monotonicity exactly once.

- if bitonic-sequence larger than block-size, then read and write directly from global memory; otherwise read/write from shared-memory
Parallel Scan (within thread-block)

**Reduction tree**

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>-2</td>
<td>0</td>
<td>1</td>
<td>-3</td>
<td>4</td>
<td>-4</td>
<td>-3</td>
</tr>
<tr>
<td>-1</td>
<td>2</td>
<td>-2</td>
<td>-2</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-4</td>
<td>-1</td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-3</td>
<td>-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Scan tree**

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-3</td>
<td>-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-3</td>
<td>-4</td>
<td>-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>-3</td>
<td>-3</td>
<td>-4</td>
<td>-5</td>
<td>-6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-3</td>
<td>1</td>
<td>-3</td>
<td>-6</td>
</tr>
</tbody>
</table>

**Construct scan tree:** right child: copy parent’s value.  
left child: difference between parent’s value and sibling’s value in reduction tree.
Libraries

Optimized libraries for

- **cuBLAS** for linear algebra
- **cuFFT** for Fast Fourier Transform
- **cuDNN** for Deep Neural Networks

*cuBLAS Demo!*
Summary

- **Calculating Occupancy**: higher is better
  - useful for debugging performance bottlenecks

- **Miscellaneous**:
  - managing multiple GPUs
  - executing multiple streams in parallel

- **Algorithms**
  - Image filtering
  - Parallel scan
  - Bitonic sort

- **Libraries**: cuBLAS, cuFFT, cuDNN