CSC266 Introduction to Parallel Computing using GPUs

GPU Architecture I (Execution)

Sreepathi Pai
October 25, 2017

URCS
Outline

- Quick Recap
- Streams and Command Queues
- GPU Execution of Kernels
- Warp Divergence
- More on GPU Occupancy
Outline

Quick Recap

Streams and Command Queues

GPU Execution of Kernels

Warp Divergence

More on GPU Occupancy
- Determine a thread block size: say, 256 threads
- Divide work by thread block size
  - Round up
  - \( \lceil N/256 \rceil \)
- Configuration can be changed every call

```c
int threads = 256;
int Nup = (N + threads - 1) / threads;
int blocks = Nup / threads;

vector_add<<<blocks, threads>>>(...)
```
Kernel Launch Configuration

- GPU kernels are SPMD kernels
  - Single-program, multiple data
  - All threads execute the same code
- Number of threads to execute is specified at launch time
  - As a grid of $B$ thread blocks of $T$ threads each
  - Total threads: $B \times T$
- Reason: Only threads within the same thread block can communicate with each other (cheaply)
  - Other reasons too, but this is the only algorithm-specific reason
Blocking and Non-blocking APIs

- **Blocking API (or operation)**
  - CPU waits for operation to finish
  - e.g. simple cudaMemcpy

- **Non-blocking API (or operation)**
  - CPU does not wait for operation to finish
  - e.g. kernel launches
  - You can wait explicitly using special CUDA APIs

- **Operations queue up**
  - Multiple kernels can be launched
  - They will execute by default in launch order
Quick Recap

Streams and Command Queues

GPU Execution of Kernels

Warp Divergence

More on GPU Occupancy
The Default Stream

- A GPU can do multiple things in parallel
  - Just like a CPU
  - Most common: overlapping memory copies and kernel executions
- Main programming construct: **Stream**
  - Purely software construct
- Stream is conceptually equivalent to a CPU thread
  - Operations in same stream happen in order
  - Operations in different streams can happen in *any* order
- Stream 0 is the *default stream*
  - All operations not on an explicit stream are on this stream
• Hardware construct
• Streams map to command queues
  • Many (streams)-to-one (hardware queue)
• About 32 hardware queues in Kepler (Hyper-Q)
Outline

Quick Recap

Streams and Command Queues

GPU Execution of Kernels

Warp Divergence

More on GPU Occupancy
Why Grids? NVIDIA: Hardware Scalability
Threads to Warps
Thread blocks to an SM
GPU Occupancy

- CPU threads share resources by time multiplexing
  - One thread owns all CPU resources (registers, etc.) for its time slice
  - Context-switches are performed by OS
- GPU threads *do not share* resources
  - Own fixed partition of resources for entire lifetime of thread
  - Context-switches are performed by hardware every few cycles
- Changing number of threads changes *utilization* of resources
## GPU Resources per SM (NVIDIA Kepler)

<table>
<thead>
<tr>
<th>Resource</th>
<th>Available</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>2048</td>
<td>1024/block</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>48K (max)</td>
<td>48K/block</td>
</tr>
<tr>
<td>Registers</td>
<td>65536</td>
<td>255/thread</td>
</tr>
<tr>
<td>Thread Blocks</td>
<td>16</td>
<td>16/SM</td>
</tr>
</tbody>
</table>

- Every block consumes:
  - $T$ threads
  - $T \times R$ registers where $R$ is registers per thread
  - 1 block
  - $SM$ shared memory per block (optional)

- The resource that gets exhausted first determines occupancy and residency
  - Occupancy: number of hardware threads utilized
  - Residency: number of hardware blocks utilized
kernel<<<2048, 32>>>()

- $T = 32$
  - thread limit $2048/32 = 64$ thread blocks
- $R = 100$ ($100 \times 32 = 3200$ per thread block)
  - register limit $65536/3200 = 20$ thread blocks
- $SM = 1K$
  - SM limit $48K/1K = 48$ thread blocks

- Limiting resource: thread blocks (16)
- Residency: 16
- Occupancy: $(16 \times 32)/2048 = 25\%$
GPU Occupancy: Example 2

\[
\text{kernel} \llll\llll<2048, 64>>\llll\llll() \llll\llll
\]

- \( T = 64 \)
  - thread limit \( 2048/64 = 32 \) thread blocks
- \( R = 100 \) (\( 100 \times 64 = 6400 \) per thread block)
  - register limit \( 65536/6400 = ? \) thread blocks
- \( SM = 1K \)
  - SM limit \( 48K/1K = 48 \) thread blocks

- Limiting resource: ?
- Residency: ?
- Occupancy: \( (? \times 64)/2048 = ? \% \)
How many threads?

- Try to maximize utilization (NVIDIA Manual)
- Later today: Better strategy
Summary

- Thread blocks are mapped to SMs "whole"
  - Atleast one thread block must fit
  - No partial thread blocks
- Up to \textit{res} thread blocks per SM
  - \textit{res} is residency
  - Different for different kernels
- Once all SMs are occupied, remaining blocks wait
  - Start running once currently running blocks finish
Outline

Quick Recap

Streams and Command Queues

GPU Execution of Kernels

Warp Divergence

More on GPU Occupancy
• All threads in a warp execute the same instruction (same PC)
• What happens when:
  • that instruction is a conditional branch?
  • is a load that misses for some threads but not others?
Divergence

- If threads in a warp decide execute different PCs, the warp splits
- Two directions for a branch
  - Two splits
  - Each split is executed serially
  - Nested branches also split correctly
- Join back at a pre-determined “meet” point
  - Immediate post-dominator
if (cond) {
    x = 1;
} else {
    y = 1;
}

• Assume warps contains four threads each
• Assume only T0, T2 have cond == true.

<table>
<thead>
<tr>
<th>Time</th>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x = 1</td>
<td>x = 1</td>
<td>x = 1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>y = 1</td>
<td></td>
<td>y = 1</td>
</tr>
</tbody>
</table>

• If cond is true for all threads

<table>
<thead>
<tr>
<th>Time</th>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x = 1</td>
<td>x = 1</td>
<td>x = 1</td>
<td>x = 1</td>
</tr>
</tbody>
</table>
Tackling Divergence

- Threads in the same warps should avoid divergent conditions
  - Easier said than done
- Threads in the same warp should try to access locations in same memory line
  - Memory divergence *repeats* requests until all threads have received data
- Compiler will predicate instructions
  - No divergence – both sides executed
  - Predicated instructions are executed but do not commit
  - Shown as [] below

<table>
<thead>
<tr>
<th>Time</th>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x = 1</td>
<td>[x = 1]</td>
<td>x = 1</td>
<td>[x = 1]</td>
</tr>
<tr>
<td>1</td>
<td>[y = 1]</td>
<td>y = 1</td>
<td>[y = 1]</td>
<td>y = 1</td>
</tr>
</tbody>
</table>
Quick Recap

Streams and Command Queues

GPU Execution of Kernels

Warp Divergence

More on GPU Occupancy
Occupancy Recap

- GPUs partition resources among running threads
- NVIDIA Manual says maximize occupancy
  - Why?
Reasoning about occupancy

```
kernel <<<x, y>>>()
```

- Consider:
  - 1 Thread Block
  - $N$ thread blocks, $N$ equal to number of SMs/SMX
  - $N \ast \text{Residency}$ thread blocks
  - $> N \ast \text{Residency}$ thread blocks
Less Occupancy?

- Is there a case to reduce occupancy/residency?
  - i.e. let threads consume more resources?
  - smaller thread blocks?
Better Performance at Lower Occupancy

### Multiplication of two large matrices, single precision (SGEMM):

<table>
<thead>
<tr>
<th></th>
<th>CUBLAS 1.1</th>
<th>CUBLAS 2.0</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads per block</td>
<td>512</td>
<td>64</td>
<td>8x smaller thread blocks</td>
</tr>
<tr>
<td>Occupancy (G80)</td>
<td>67%</td>
<td>33%</td>
<td>2x lower occupancy</td>
</tr>
<tr>
<td>Performance (G80)</td>
<td>128 Gflop/s</td>
<td>204 Gflop/s</td>
<td>1.6x higher performance</td>
</tr>
</tbody>
</table>

### Batch of 1024-point complex-to-complex FFTs, single precision:

<table>
<thead>
<tr>
<th></th>
<th>CUFFT 2.2</th>
<th>CUFFT 2.3</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads per block</td>
<td>256</td>
<td>64</td>
<td>4x smaller thread blocks</td>
</tr>
<tr>
<td>Occupancy (G80)</td>
<td>33%</td>
<td>17%</td>
<td>2x lower occupancy</td>
</tr>
<tr>
<td>Performance (G80)</td>
<td>45 Gflop/s</td>
<td>93 Gflop/s</td>
<td>2x higher performance</td>
</tr>
</tbody>
</table>

Volkov, V., “Better Performance at Lower Occupancy”, GTC 2010
Volkov’s Insights

- Do more parallel work per thread to hide latency with fewer threads (i.e. increase ILP)
  - Unroll
- Use more registers per thread to access slower shared memory less
  - Shared memory latency comparable to registers, but
  - Shared memory throughput is lower!
- Both may be accomplished by computing multiple outputs per thread
- Note that Volkov underutilizes threads, but maxes out registers!
  - Fermi had 63 registers/thread, Kepler has 255 registers/thread
  - Why have a register limit?