CUDA Programming Model

Ming Yang
Apr 5, 2016
• What are the scheduling units in Streaming Multiprocessor (SM)??

• warps. How are they scheduled?

• How is the occupancy computed??

• anything to do with block/thread/registers/shared memory? Yes! All of them.
dim3 dimGrid(2, 2, 1);
dim3 dimBlock(4, 2, 1);
vectorAdd<<<dimGrid, dimBlock>>>(a, b, c);

How are these threads assigned to the SMs??

# of threads: 
(2*2) * (4*2) = 32
Thread Blocks Assignment

- **Threads** are assigned to **SM** in **block** granularity.
- **Blocks** in one **grid** can be assigned to different **SMs**.
- **SM** manages/schedules **thread** execution.
  - *how??*
Warps as Scheduling Units

- Each block is executed as 32-thread warps
- Warps are scheduling units in SM
  - how are they scheduled?
- Threads in a warp execute in SIMT
  - what is SIMT (Single Instruction Multiple Thread)?
  - What about control divergence?
Warps as Scheduling Units (cont.)

- Warps are scheduling units in SM
Warps as Scheduling Units (cont.)

- Threads in a warp execute in SIMT
Review

- **Threads** are organized by **block/grid**
- **Threads** are assigned to **SM** in **block** granularity
- **Threads** are scheduled in the unit of warp, and in the way of **SIMD**
Occupancy

- Occupancy = # of active warps / Maximum number of resident warps per SM

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>Compute Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.x</td>
</tr>
<tr>
<td>Maximum number of resident warps per SM</td>
<td>48</td>
</tr>
</tbody>
</table>

- Occupancy limiters:
  - Register usage
  - Shared memory usage
  - Block size
Memory hierarchy

Per-thread local memory

Per-block shared memory

Global Memory
Occupancy limiter: Register usage

Example 1 (capability = 3.0)

- Kernel uses 21 registers per thread
- # of active threads = 64K / 21 \(\approx\) 3121
  - > 2048 thus an occupancy of 100%

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>Compute Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of 32-bit registers per thread block</td>
<td>2.x</td>
</tr>
<tr>
<td>Maximum number of resident threads per SM</td>
<td>3.0</td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread block</td>
<td>3.2</td>
</tr>
<tr>
<td>Maximum number of resident threads per SM</td>
<td>3.5</td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread block</td>
<td>3.7</td>
</tr>
<tr>
<td>Maximum number of resident threads per SM</td>
<td>5.0</td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread block</td>
<td>5.2</td>
</tr>
<tr>
<td>Maximum number of resident threads per SM</td>
<td>5.3</td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread block</td>
<td>32 K</td>
</tr>
<tr>
<td>Maximum number of resident threads per SM</td>
<td>64 K</td>
</tr>
<tr>
<td>Maximum number of resident threads per SM</td>
<td>32 K</td>
</tr>
<tr>
<td>Maximum number of resident threads per SM</td>
<td>1536</td>
</tr>
<tr>
<td>Maximum number of resident threads per SM</td>
<td>2048</td>
</tr>
</tbody>
</table>
Occupancy limiter: Register usage (cont.)

- Example 2 (capability = 3.0)
  - Kernel uses 64 registers per thread
  - \# of Active threads = 64K / 64 = 1024
    - \# of warps = 1024 / 32 = 32
    - Occupancy = 32 / 64 = 50%
Occupancy limiter: Shared memory

Example 1 (capability = 3.0)

- Kernel uses 16 bytes of shared memory per thread
- # of Active threads = 48K / 16 = 3072
- > 2048 thus an occupancy of 100%

### Technical Specifications

<table>
<thead>
<tr>
<th></th>
<th>Compute Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.x  3.0  3.2  3.5  3.7  5.0  5.2  5.3</td>
</tr>
<tr>
<td>Maximum amount of shared memory per SM</td>
<td>48 KB 112 KB 64 KB 96 KB 64 KB</td>
</tr>
<tr>
<td>Maximum number of resident threads per SM</td>
<td>1536 2048</td>
</tr>
<tr>
<td>Maximum number of resident warps per SM</td>
<td>48 64</td>
</tr>
</tbody>
</table>
Occupancy limiter: Shared memory (cont.)

• Example 2 (capability = 3.0)
  • Kernel uses 32 bytes of shared memory per thread
  • # of Active threads = 48K / 32 = 1536
    • # of warps = 1536 / 32 = 48
    • Occupancy = 48 / 64 = 75%

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>Compute Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.x</td>
</tr>
<tr>
<td>Maximum amount of shared memory per SM</td>
<td>48 KB</td>
</tr>
<tr>
<td>Maximum number of resident threads per SM</td>
<td>1536</td>
</tr>
<tr>
<td>Maximum number of resident warps per SM</td>
<td>48</td>
</tr>
</tbody>
</table>
# Occupancy limiter: Block size

## Technical Specifications

<table>
<thead>
<tr>
<th>Maximum number of resident blocks per multiprocessor</th>
<th>Compute Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of resident threads per SM</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Maximum number of resident warps per SM</td>
<td>1536</td>
</tr>
<tr>
<td></td>
<td>48</td>
</tr>
</tbody>
</table>

- **capability = 3.0**

<table>
<thead>
<tr>
<th>Block size</th>
<th>Active threads</th>
<th>Active warps</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>32 * 16 = 512</td>
<td>512 / 32 = 16</td>
<td>16 / 64 = 25%</td>
</tr>
<tr>
<td>64</td>
<td>1024</td>
<td>32</td>
<td>50%</td>
</tr>
<tr>
<td>128</td>
<td>2048</td>
<td>64</td>
<td>100%</td>
</tr>
<tr>
<td>192</td>
<td>3072 (2048)</td>
<td>64</td>
<td>100%</td>
</tr>
<tr>
<td>256</td>
<td>4096 (2048)</td>
<td>64</td>
<td>100%</td>
</tr>
</tbody>
</table>

Warp size = 32
Occupancy

• Do we want higher occupancy?
  • Maybe yes. Latency (of memory op. and algorithmic op.) can be hidden with more threads running.

• Is occupancy a metric of performance?
  • No!! It’s just one of the contributing factors.
fixed instruction parallelism (ILP=1)

fixed thread parallelism (12.5% occupancy)

Reference:
http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf
Review

- Calculation formula for occupancy
  - \# of active warps / maximum number of warps per SM
- Occupancy limiters:
  - register, shared memory, block size
- Understanding of occupancy
  - occupancy is not equivalent to performance
  - but we still want higher occupancy usually
Case study: cublasSgemm

- **Matrix multiplication** of single-precision real number
- **SGEMM** performs one of the matrix-matrix operations
  - \( C := \alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot C \)
  - where \( \text{op}(X) \) is one of
    - \( \text{op}(X) = X \)
    - \( \text{op}(X) = X^\top \) (transposed)
  - \( \alpha = 1.0 \)
  - \( \beta = 0.0 \)
- It’s used by the fully-connected (fc) layer in Caffe (when batch size is larger than 1)
Reasons of case-studying cublasSgemm

- **sgemm_largek_lds64**
  - it’s the kernel used by cublasSgemm
  - it decreases fastest with batch size increasing
  - it’s the only kernel I observed of which occupancy changes with different batch sizes
Experiment

• Use cublasSgemm:
  • Inputs: Matrix A (M*K), B (K*N)
  • Output: Matrix C (M*N) = A*B

• Variables used here are consistent with the usage in the fully-connected layer in Caffe)
  • M: batch size (2, 4, 8, …, 1024)
  • K: 9216/4096/4096
  • N: 4096/4096/1000
Results

Execution time (ms.)

Occupancy

sgemm_largek_lds64

maxwell_sgemm_128x128_nn

maxwell_sgemm_128x64_nn

sgemm_largek_lds64 with different parameters

Batch size (M)

- 2: 53.3
- 4: 33.5
- 8: 40.5
- 16: 96.6
- 32: 97.2
- 64: 64.6
- 128: 135.1
- 256: 269.5
- 512: 542.6
- 1024: 1,085.0

Execution time (ms.)

- 0
- 0.25
- 0.5
- 0.75
- 1

Batch size (M)

- 2
- 4
- 8
- 16
- 32
- 64
- 128
- 256
- 512
- 1024

sgemm_largek_lds64

<<<64*1*8, 32*4*1>>>

maxwell_sgemm_128x128_nn

<<<32*1*1, 256*1*1>>>

maxwell_sgemm_128x64_nn

<<<32*1*1, 128*1*1>>>

sgemm_largek_lds64

<<<128*1*4, 16*16*1>>>
Summary

• Thread hierarchy
• Streaming multiprocessor scheduling
• Memory hierarchy
• Occupancy
• Case study on `cublasSgemm`
References

- (Coursera class) Heterogeneous Parallel Programming by Wen-mei W. Hwu (https://class.coursera.org/hetero-004)
Backup slides (about stream and concurrency) after this

They’re basically copied from
Streams

- A sequence of operations that execute in issue-order on the GPU
- Programming model used to effect concurrency
  - CUDA operations in different streams may run concurrently
  - CUDA operations from different streams may be interleaved
- Rules:
  - A CUDA operation is dispatched from the engine queue if:
    - Preceding calls in the same stream have completed,
    - Preceding calls in the same queue have been dispatched, and
    - Resources are available
Example

Two streams, stream 1 is issued first
- **Stream 1**: HDa1, HDb1, K1, DH1 (issued first)
- **Stream 2**: DH2 (completely independent of stream 1)

CUDA operations get added to queues in issue order
- issue order
  - HDa1
  - HDb1
  - K1
  - DH1
  - DH2

Signals between queues enforce synchronization

within queues, stream dependencies are lost

DH1 blocks completely independent DH2

execution
- time
  - K1
  - DH1
  - DH2

runtime = 5
Example

Two streams, stream 2 is issued first

- Stream 1: HDa1, HDb1, K1, DH1
- Stream 2: DH2 (issued first)

CUDA operations get added to queues in issue order

Signals between queues enforce synchronization

within queues, stream dependencies are lost

issue order matters!

program

<table>
<thead>
<tr>
<th>issue order</th>
</tr>
</thead>
<tbody>
<tr>
<td>DH2</td>
</tr>
<tr>
<td>HDa1</td>
</tr>
<tr>
<td>HDb1</td>
</tr>
<tr>
<td>K1</td>
</tr>
<tr>
<td>DH1</td>
</tr>
</tbody>
</table>

H2D queue

<table>
<thead>
<tr>
<th>H2D queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDa1</td>
</tr>
<tr>
<td>HDb1</td>
</tr>
</tbody>
</table>

compute queue

<table>
<thead>
<tr>
<th>compute queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
</tr>
</tbody>
</table>

D2H queue

<table>
<thead>
<tr>
<th>D2H queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>DH2</td>
</tr>
<tr>
<td>DH1</td>
</tr>
</tbody>
</table>

execution

<table>
<thead>
<tr>
<th>execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDa1</td>
</tr>
<tr>
<td>HDb1</td>
</tr>
</tbody>
</table>

concurrent

<table>
<thead>
<tr>
<th>concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>DH2</td>
</tr>
<tr>
<td>K1</td>
</tr>
<tr>
<td>DH1</td>
</tr>
</tbody>
</table>

runtime = 4