CUDA Threads
Bedřich Beneš, Ph.D.
Purdue University
Department of Computer Graphics

Terminology

- **Streaming Multiprocessor (SM)**

- **Streaming Processors (SP)**
  also called **CUDA Cores**

- A SP processes threads belonging to a block (shared resources)

---

How it works

1) Grid is launched

2) Blocks are assigned to streaming multiprocessors (SM) on block-by-block basis in arbitrary order (scalability)
  (Each SM can process more blocks)

  e.g., GT200 can do max 8 blocks or max 1024 threads per SM
How it works

3) An assigned block is partitioned into warps. Their execution is interleaved.

4) Warps are assigned to SM (one thread to one SP)

5) Warps can be delayed if idle for some reason (waiting for memory)

Basic Considerations

- the size of a block is limited to 512 threads
  blockDim(512,1,1)
  blockDim(8,16,2)
  blockDim(16,16,2)

- kernel can handle up to 65,536x65,536 blocks

G80 Architecture

has **16 SMs**
each can process
≤ 8 blocks
or
≤ 768 threads
max: 8x16=128 CUDA Cores (SPs)
max: 16x768=12,288 threads

GT200 Architecture

has **30 SMs**
each can process
≤ 8 blocks
or
≤ 1024 threads
max: 8x30=240 CUDA Cores (SPs)
max: 30x1,024= 30,720 threads
GT200 Architecture

30,720 threads max
240 CUDA cores
One SM limits:
1024 threads = 4x256 or 8x128 etc.
One block limits:
512 threads = 2x256 or 8x64 etc.

GT400 (Fermi)

has 16 SM
each can process
≤ 8 blocks
1 SM has 32 cuda cores
total: 512 cuda cores
plus 16kb or 48kb L1 Caches per SM
can run two different warps per kernel
(dual warp scheduler)

Block Assignment

• if more than the max amount of blocks are assigned to SM
  they will be scheduled for later execution

• Is it good or bad?
  Well, it depends, but usually good.
### Comparison

<table>
<thead>
<tr>
<th></th>
<th>G80</th>
<th>GT200</th>
<th>GT500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistors</td>
<td>6.8T</td>
<td>1.4T</td>
<td>3.0T</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>128</td>
<td>240</td>
<td>512</td>
</tr>
<tr>
<td>Warp scheduler per SM</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Shard Memory per SM</td>
<td>16kB</td>
<td>16kB</td>
<td>16 or 48kB</td>
</tr>
<tr>
<td>L1 cache per SM</td>
<td>None</td>
<td>None</td>
<td>16 or 48kB</td>
</tr>
<tr>
<td>L2 cache per SM</td>
<td>None</td>
<td>None</td>
<td>768 kB</td>
</tr>
<tr>
<td>Load/store address width</td>
<td>32b</td>
<td>32b</td>
<td>64b</td>
</tr>
</tbody>
</table>

### Warps

- A thread block is divided into warps
- A block of 32 threads (hw dependent and can change)
- **Warps are scheduling units of SM**
- \( \text{warp}_0: t_0, t_1, \ldots, t_{31} \)
- \( \text{warp}_1: t_{32}, t_{33}, \ldots, t_{63} \)

### Warps

- **Example:**
  - 3 blocks assigned to SM, each with 128 threads.
  - *How many warps we have in the SM?*
  - 128 threads/32 (warp length) = 4 warps
  - 4(warps) x 3 (blocks) = 12 warps at the same time

- **Example2:**
  - *How many warps in the GT200?*
  - 1024 threads/32 (warp length) = 32 warps
Warp Assignment

- one thread is assigned to one SP
- SM has 8 SPs
- warp has 32 threads
- so **a warp is executed in four steps**

Warps – latency hiding

- Why do we need so many warps if there are just 8 CUDA cores in SM (GT200)?
  
  **Latency hiding:**
  - a warp executes a global memory read instruction that delays it for 400 cycles
  - any *other warp* can be executed in the meantime
  - if more than one is available - priorities

Warps – processing

- A warp is SIMT
  (single instruction multiple thread)
  all run in parallel and the same instruction
- Two warps are MIMD
  can do branching, loops, etc.
- Threads within one warp do not need synchronization – they run the same time instruction

Warps – zero-overhead

**Zero-overhead thread scheduling**

- having many warps available, the selection of warps that are ready to go keeps the SM busy (no idle time)
- that is why, caches are not usually necessary
Example - granularity

- Having GT200 and matrix multiplication. Which tiles are the best 4x4, 8x8, 16x16, or 32x32?

Example - granularity

- 4x4 will need 16 threads per block
  - SM can take up to 1024 threads
  - We can take 1024/16=64 blocks
  - BUT! The SM is limited to 8 blocks
  - There will be 8*16=128 threads in each SM
  - 128/32=4 -> 8 warps, but each half full
  - heavily underutilized!
  - (fewer warps to schedule)

Example - granularity

- 8x8 will need 64 threads per block
  - SM can take up to 1024 threads
  - We can take 1024/64=16 blocks
  - BUT! The SM is limited to 8 blocks
  - There will be 8*64=512 threads in each SM
  - 512/32=16 warps
  - still underutilized!
  - (fewer warps to schedule)

Example - granularity

- 16x16 will need 256 threads per block
  - SM can take up to 1024 threads
  - We can take 1024/256=4 blocks
  - The SM can take it 2x
  - There will be 8*64=512 threads in each SM
  - 512/32=16
  - full capacity and a lot of warps to schedule
Example - granularity

- 32x32 will need 1024 threads per block a block (GT200) can take max 512
  Not even one will fit in the SM
  (not true in GT400)

Example - granularity

- granularity does not automatically mean a good performance
- depends on using shared memory, branching, loops, etc.
- but it does mean low latency
- Blocks (resp. # of threads in block) should be multiples 32 for better alignment

Warps/block alignment

- **1D Case**
  block of 100 threads – how many warps?
  100/32=3+1/4
  
  \[
  \begin{array}{cccc}
  t_0 & t_1 & \ldots & t_{31} \\
  & t_{32} & & t_{63} \\
  & & t_{64} & \ldots \\
  & & & t_{94} \\
  w_0 & w_1 & w_2 & \frac{1}{4}\text{ of } w_3 \\
  \end{array}
  \]

  - the last warp will be occupied entirely, but only the 8 threads will have meaning

Warps/block alignment

- **2D Case**
  blockDim(9,9)
  81 threads
  100/32=2 warps and 17 threads

  \[
  \begin{array}{cccc}
  & & & \\
  & & t_{0,0} & \ldots \\
  & t_{0,1} & \ldots & t_{8,1} \\
  & t_{0,2} & \ldots & t_{8,2} \\
  w_0(32) & w_1(32) & w_3(17) \\
  \end{array}
  \]
**Warp execution**

- **SIMT** – single instruction, multiple threads. The same instruction is broadcasted to all threads and execute at the same time in the SM.
- All SPs in the SM execute the same instruction.

---

**Thread Divergence**

- How can all threads execute the same instruction if we have the “if” command?

*Example:*
```
if (threadIdx.x<10)
    {a[0]=10;}
else {a[1]=10;}
```

Threads [0-9] will do “then”
the others will do “else”
This is called **thread divergence**

---

**Example:**
```
a=tex2D(tex,u,v);
if (a<0.5)
    {a[0]=10;}
else {a[1]=10;}
```
Thread Divergence

- What causes thread divergence?

1) If statements with functions of threadIdx
2) Loops with functions of threadIdx

if statements are expensive anyway…

Example:
for (int i=0;i<threadIdx.x;i++)
a[i]=i;

All loops that should finished will finish, but
the GPU will iterate for the others till the end

Reading

- NVIDIA CUDA Programming Guide