Warps and blocks

• Threads in GPUs are organized in two ways:
  • *warps* (always 32 threads)
  • *blocks* (you can choose the number of threads)

• Why do we need these concepts?
  • warps: help the *hardware*
  • blocks: help the *programmer*
What if there were no warps?

• Our GPU has 640 arithmetic units for doing scalar operations

• If we had only individual threads, you would need **640 schedulers** that process instructions from individual threads and move operands to the right arithmetic units

• By organizing threads in warps, we only need **20 schedulers** that process instructions from complete warps and move warp-wide operands to warp-wide arithmetic units

• Less space & energy used by control logic, more space & energy left for useful work
What if there were no warps?

• Similar to CPUs & vector operations:
  • make *arithmetic units and registers wider* by a factor of 8: more processing power without adding much more control logic
  • *increase the number of cores* by a factor of 8: everything got 8 times more costly
Blocks and shared memory

• Blocks are there to help you!

• You can allocate a small amount of very fast “shared memory” for each block
  • “small amount” ≈ kilobytes per block
  • “very fast” ≈ L1 cache

• All threads of a block see the same shared memory

• Threads of a block can use shared memory to communicate with each other and coordinate their work
Blocks and shared memory

- **Example**: each block calculates a sum using many threads
  - $b$ threads per block
  - allocate $b$ words of shared memory per block
  - split input in $b$ parts
  - thread $i$ calculates a local sum in its own part and *writes* it in element $i$ in shared memory
  - **synchronization**: wait all threads to finish writing
  - thread 0 *reads* all local sums from shared memory and calculates the grand total
Warps and blocks

• **Warps:**
  - always 32 threads
  - helps with hardware design:
    lots of arithmetic power with a simpler control
  - you will have to live with this even if it is inconvenient for you

• **Blocks:**
  - you can choose the block size (e.g. 64 or 256 threads)
  - threads of a block can easily and efficiently communicate with each other using “shared memory”
  - blocks are a useful feature that you can use
Using shared memory in CUDA

```c
__global__ void mykernel() {
    __shared__ float x[100];
    ...
}
```

One array per block!

Different:
- x[0] in block 10
- x[0] in block 11

Same:
- x[0] in thread 5 of block 10
- x[0] in thread 6 of block 10
Using shared memory in CUDA

```c
__global__ void mykernel() {
    __shared__ float x[100];
    ...
    __syncthreads();
    ...
}
```

A thread won't continue until all threads of the block have reached this point.
Using shared memory in CUDA

... Write to my own slot

\[ x[i] = a; \]
Using shared memory in CUDA

\[ x[i] = a; \]

\_\_syncthreads();

Write to my own slot

Wait for everyone to finish writing
Using shared memory in CUDA

\[ x[i] = a; \]
\[ _syncthreads(); \]
\[ b = x[0]; \]

- Write to my own slot
- Wait for everyone to finish writing
- Now safe to read from any slot
Using shared memory in CUDA

... Write to my own slot

\[ x[i] = a; \]

\_\_syncthreads();

Wait for everyone to finish writing

b = x[0];

Now safe to read from any slot

\_\_syncthreads();

Wait for everyone to finish reading
Using shared memory in CUDA

... Write to my own slot

x[i] = a;
__syncthreads();
b = x[0];
__syncthreads();
x[i] = c;
...

Wait for everyone to finish writing

Now safe to read from any slot

Wait for everyone to finish reading

Now safe to write again
Shared memory is small

• **64 KB** in total per SM (“streaming multiprocessor”)

• Example:
  • you want to have 8 active blocks on each SM
  • you can only allocate at most 8 KB of shared memory per block
Key elements of CUDA programs

• Allocating *GPU memory*, moving data between CPU memory and GPU memory

• Creating *blocks of threads*, launching *kernels*

• Allocating *shared memory* for sharing data inside a block, using `__syncthreads` to synchronize work