SOFT354 - Increasing performance of CUDA Programs

Date: 13-10-16

2D Blocks and Grids

Remember: - A block consists of multiple **threads - a grid** consists of multiple **blocks**

- Number of blocks to use depends on what you're trying to process
- Use of expierementation to get what is right for you
  - Make code function without dependancy on how many blocks it uses
- Use dim3 objects to lauch kernel with grid/block with dimensions greater than one

• To launch a kernel with grid / block dimensions that aren't one dimensional use **dim3** objects.

```
dim3 gridSize(94, 125, 1);
dim3 blockSize(32, 32, 1);
invertKernel<<<gridSize, blockSize>>>();
```

Need to give z dimensions as well – just set to 1.

• Kernel can now determine its global position in the image using **blockIdx**, **blockDim** and **threadIdx**:

```
__global__ void invertKernel() {
   int globalIdX = blockIdx.x * blockDim.x + threadIdx.x;
   int globalIdY = blockIdx.y * blockDim.y + threadIdx.y;
   if (globalIdX < 2992 && globalIdY < 4000) {
      // Do something...
   }
}
```

- Use globalId of x and y to get the position of the threads calculation within the grid as a whole - Clause in if statement checking the global id gets rid of threads that don't have a thing to process if you had to round up on number of blocks

- Can do the same but include z direction with 3D blocks
- Z dimensional isn't as high as X or Y
  - X/Y = 1024
  - Z = 64

```
dim3 gridSize(100, 100, 100);
dim3 blockSize(10, 10, 10);
someKernel<<<gridSize, blockSize>>>();
```

```
__global__ void someKernel() {
   int globalIdX = blockIdx.x * blockDim.x + threadIdx.x;
   int globalIdY = blockIdx.y * blockDim.y + threadIdx.y;
   int globalIdZ = blockIdx.z * blockDim.z + threadIdx.z;
```

Linearisation of blocks and grids

- 2D and 3D blocks and grids are just for convenience
- CUDA will convert you **blocks** into a 1D array of **threads**
  - .. and your **grid** into a 1D array of **blocks**

Block assignment

- A GPU has multiple *streamng multiprocessors* (SMs)
- Each SM has one or more blocks assigned to it (“**resident**”) at any given time
- Only **whole blocks** can be assigned to SMs, therefore **all threads in a block will run on the same SM**
  - Important for memory sharing

Warps

- When a streaming multiprocessor has had some blocks assigned to it
- when it takes the block, it breaks it down into warps
- **warps** are a set of 32 **threads**
  - These will be executed in parallel

Warp Scheduling

- Generally you don’t need to worry about warps
- But by understanding how an SM shcedules warps you can improve performance
- Only one warp can be running at any given time
- Any given time a warp can be in one of three states:
  1. **Running** currently being executed by the SM
  2. **Waiting** Can’t run because it’s waitng on something - usually some data to read from memory
  3. **Ready** Not waiting but is ready to run once it is its turn

- Ideal situation is when there is one currently running, two are ready to run when it is finished
- Bad situation is if they're all waiting
- If all warps were waiting, none could be executed and therefore they are **stalled**
  - Usually waiting for memory access

Reducing stalls

- The most effective way to reduce stalls (assuming they're mostly caused by memory access) is to use memory more effectively
  - Coalesced memory reads and shared memory
- But it is also important to maximise the nymber of warps resident in an SM (its **occupancy**);
- The more resident warps, the more chance there's one **ready** to be executed

Occupancy

- When looking at how many warps can be resident in a SM
- Points to consider
  - Max no of threads per block
  - Max no of resident blocks per multiprocessor
  - Max no of resident warps per multiprocessor
  - Max no of resident threads per multiprocessor

- 8x8 blocks:
  - 64 threads per block (2 warps).
  - But each SM can only contain up to 8 blocks.
  - So maximum number of resident threads is 8*64=512.
  - Much less than the capacity (1536 threads).

- 16x16 blocks:
  - 256 threads per block (8 warps)
  - If SM contains 6 blocks, 6*256=1536 threads.
  - Maximum possible occupancy – very good!

- 32x32 blocks:
  - 1024 threads per block (32 warps).
  - Maximum capacity is 1536 threads.
  - But can only allocate whole blocks to a SM, so there's only space for one.
  - So maximum number of resident threads is 1024.
  - Much less than the capacity (1536 threads).

- There are limits to how many registers and shared memory a SM has - This can also limit occupancy

Coalesced Memory access

- When memory being accessed is within 128 bytes of each other
  - You can access the all memory withing that range
- This saves optimistation as you don't have to multiple calls to the memory

Shared memory

- We've already met two types of statically allocated device memory:
  - **device** variables are stored in **global memory** which is *huge but slow*
  - **constant** vairables are cached in the **SM**; can be read by all threads *in a single cycle* but are *read-only*
- The last to consider is **shared**
  - How the data should be put together
- A variable with the **shared** modifier has its memory allocated in the SM's shared memory area
- This is **fast** but local to the SM
- Each block has a separate copy of any **shared** variables
- All threads in a block can access their block's copy of the shared variables, but not other blocks' copies
- Shared memory can be used to allow threads **within a block** to communicate with each other very quickly
- Common pattern is when all threads in a block need the same chunk of global memory:
  - Each thread loads a bit of the global data into shared memory (happens in parallel)
  - *Library metaphor:* Study group makes a list of books they need, everyone fetches a book from gloabl memory (shelves) and puts it on a shared desk

___syncthreads()

- Function that can only be called from the kernel code
- No gaurentuee if threads have done their job without ___syncthreads()
- Causes all threads in the current block to wait, until they've all reached the call

Conclusions

- **Threads** are organised into **blocks**, which are organised into a **grid**
- The **threads** in a **block** and/or the **blocks** in a **grid** can be organised as 1D, 2D or 3D structures
- Whatever the dimensionality, **blocks** and **threads** are linearised into a big 1D array of threads
- Whole **blocks** are allocated ot streaming multiprocessors as they become available
- When a block is allocated to a multiprocessor it is divided into **warps** of 32 threads
- Want to avoid the situation where all **warps** in a SM are waiting and not runnable, Two strategies:
  - Maximise the number of warps in an SM (its *occupancy*) by optimising block size
  - Reduce the time spent waiting for memory access by using **coalescing** and **shared memory**