NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4200 Grand Summary, pt.2

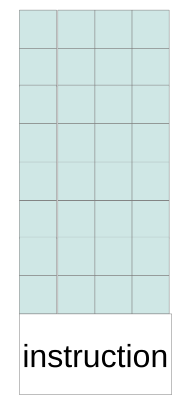
Jan.Christian.Meyer@ntnu.no

GPU architecture

- The processing unit is the “Streaming Multiprocessor” (SM)
  - 50-to-100ish in number per GPU
- Each SM contains 4 banks of 16-32 arithmetic units

  (...depending on instruction, but 32 floating point units for sure)
- These don’t have individual instruction decoders

  (...so they’re all expected to do the same thing)
- 32 CUDA threads in a group that is expected to share each instruction is called a “warp”
  - These are the GPU’s scheduling units

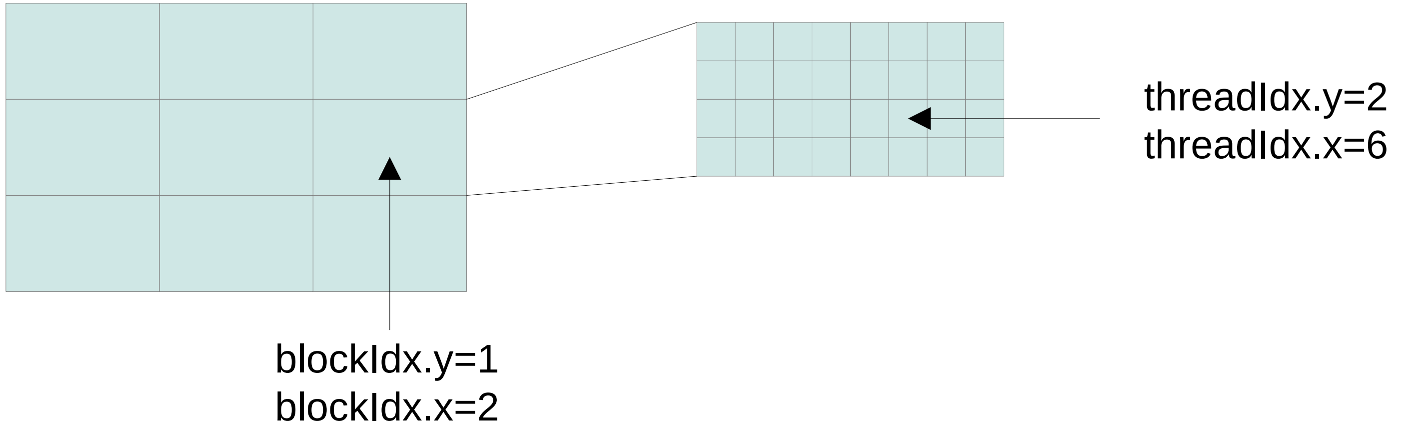


CUDA threads/cores

- NVidia likes to call each arithmetic unit a “CUDA core”
  - We can’t really stop them, it’s their trademark
- They also like to call the work that is assigned to each one a “CUDA thread”
  - See comment above
- The combination of warps and 32-wide ALU blocks have more in common with other definitions of thread and core
  - It combines an instruction pointer and some vector operations
- Personally, I think they just like big numbers
  - “16384 CUDA cores” sounds cooler than “128 processors”

Processing model

- At the top level, threads are arranged in a *grid* of *thread blocks*



- The illustration assumes a 3x3 grid of 4x8 blocks
  - The actual setup is configurable
  - Also available in 1D and 3D
  - Limited to x*y*z <= 1024 and z<= 64, though

NTNU – Trondheim
Norwegian University of Science and Technology

Programming model

- CUDA C++ is a (proprietary) extension of C++
  - It adds a few non-standard bits and bobs to the syntax
  - It needs to generate object code for two different architectures: the host (regular CPU) and the device (GPU)
  - Therefore, it needs a dedicated compiler (‘nvcc’)
- Names declared with…

  __host__ only go in the host code’s name table

  __device__ only go in the device code’s name table

  __global__ goes in both, making device functions callable from the host code

NTNU – Trondheim
Norwegian University of Science and Technology

Calling GPU functions

- Device functions are called 'kernels'

  (like so many other things)

- We have a type for specifying grid and block sizes
  - dim3 someSize(16,2); ← *1,2,3D coord. space size*
- Kernel invocation:

  my_kernel <<< gridSize, blockSize >>> (arg1, arg2);

  starts

  __global__ void my_kernel (float arg1, int arg2);

  on the graphics processor

- If you only want 1D sizes, you can just use scalars instead of dim3-s

Inside the kernel

- The kernel is invoked in (gridSize x blockSize) instances
- The variables blockIdx.{x,y,z} and threadIdx.{x,y,z} contain the x/y/z coordinates of each invocation, thus allowing us to distinguish them from each other
  - (and thus, make them work on separate data elements)
- Technically, we *can* use the coordinates in the all the same ways as ‘rank’ or ‘tid’ variables for processes and threads… **BUT:**

Warp divergence

- The thread blocks are scheduled onto the warp-capable SM units

  (Aside: this means they work best when they have a size that multiplies to 32, but it's not a requirement)

- They all run the same instruction at any given time, vector-style
- If threads within a warp take different branches of a conditional, things slow down

Warp divergence

Pretend we're in 1D, for simplicity

```
__device__ float diverging_kernel ( *results ) {
    int my_id = threadIdx.x;
    // Both of these branches will run on all threads
    if ( ( my_id % 2 ) == 0 ) {
        results[my_id] = 3.14;   // Only even-index ids work, odds idle
    } else {
        results[my_id] = 2.71;   // Only odd-index ids work, evens idle
    }
}
```

- This gets us ½ the speed compared to execution with individual instruction pointers

Synchronization

- There's a barrier called __syncthreads()
- It only synchronizes threads in the same block
- ~~To the best of my knowledge, the only way to synchronize the whole grid is to end the kernel function and make the host wait for its completion~~
  - Collaborative groups admit global synch. since CUDA 9

Memory

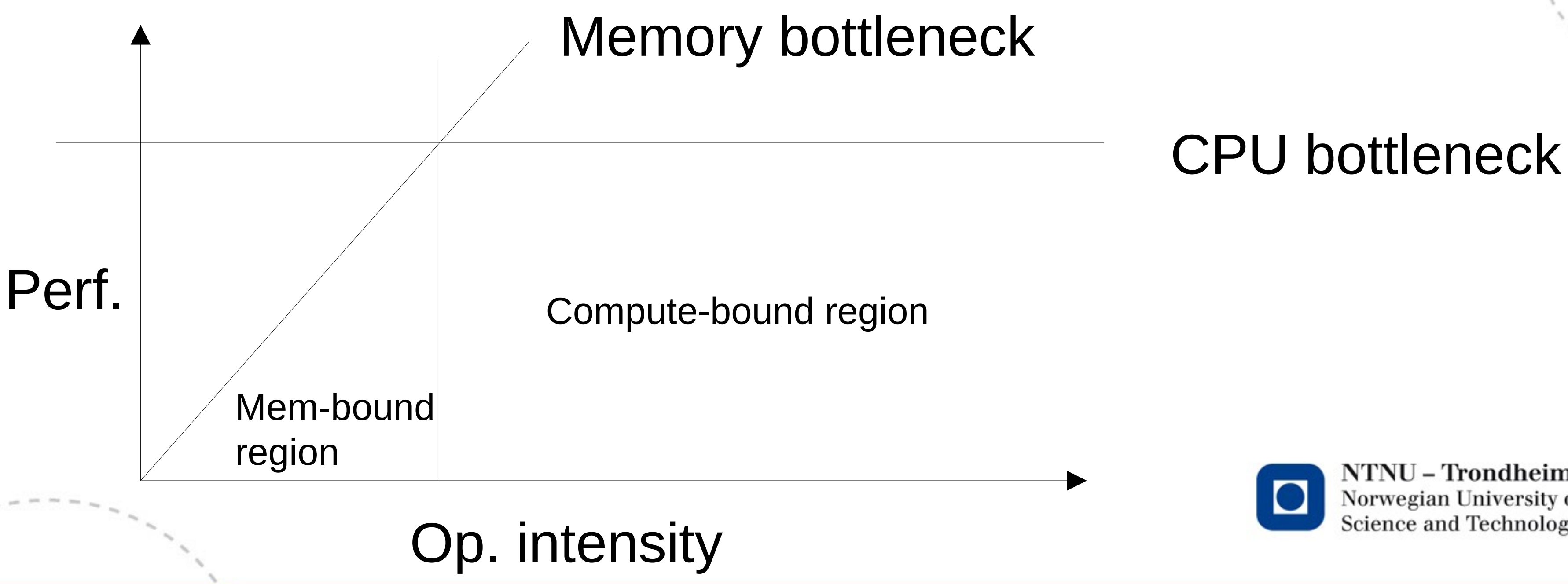
- Ideally, kernel-local variables fit in a register file on the SM
- If they don't, there's a small amount of additional memory to spill register values into
  - This does the job of the run-time stack on the host CPU
- There's a large, global memory that all threads in the grid can use
  - That is your card's video RAM
- There are smaller, faster local memories associated with the SMs
  - Variables declared as __shared__ go here
  - Kind of like cache from a chip-design point of view, but it's explicitly programmed, so I'm inclined to call it a *scratchpad* instead

The roofline model

- Estimates whether a program's performance is restricted by memory speed or compute operations
- Graphical model
  - X axis is arithmetic/operational intensity
  - Y axis is operations per second
- Obtaining op. intensity:
  - Count operations and data elements they apply to in the program

The roofline model

- Inverse memory bandwidth (in seconds/byte) is gradient of diagonal line
  - Measure, or find it in the computer spec. sheet
- Peak operations rate is level of horizontal line
  - Measure, or find it in the computer spec. sheet



High-level overview

- Programming models
  - MPI
  - Pthreads
  - OpenMP
  - Vector intrinsics
  - CUDA
- Performance models
  - Amdahl (strong scaling)
  - Gustafson (weak scaling)
  - Hockney (communication)
  - Roofline (local computing speed)

High-level overview

- Architectual elements
  - von Neumann model
  - Cache memory
  - Cache coherence (snooping / directory)
  - ILP: pipelining, OO execution, prefetching/branch pred., vectors
  - Shared vs. Distributed memory systems
  - GPU/SIMT execution (as seen from CUDA)

(Problem models:

We've had a quick look at how to solve differential equations with Finite Difference methods)

That’s it from me

- You should now be equipped to design programs for (almost) any size of parallel computer
  - We only had time for 1 method per system class, but that’s ok
- I hope you’ve had some fun between the segmentation faults

- Don’t hesitate to contact me if you want to talk about HPC in any other context
  - I’m always interested in this stuff, you don’t have to be taking a class