CMSC 416 Scribe Class Thursday 3-28

CUDA GPU Programming

CUDA is a software library used for Nvidia GPUs. It is used to accelerate computation. CUDA serves as a wrapper around the C++ language, supporting C++ and adding more features. It lets you program for Nvidia GPUs.

Compiling CUDA:
It uses the NVCC compiler. It requires the "generate-code" flag. You can look up specific compute codes for your GPU, and the compiler will generate the optimal code for the hardware.

Saxpy Kernel:
The _global_ keyword signifies a kernel runnable on the GPU.
threadIdx.x tells you which thread you are running on.
When the code is executed, each thread will launch and run the code.
Triple angle bracket syntax indicates that the code should run on the GPU.

Pitfalls when running real-world code:
N > 1024: This is a software limitation of 1024 threads per block. You will get an error if you try to launch with more.
N > # device threads: If you don't have enough device threads for all of your elements, it is also a problem.
We want to design CUDA code such that it can handle both issues.

Multiple Blocks:
Each block has up to 1024 threads.
You can specify the number of threads per block.
It's possible to launch on multiple blocks.
To ensure you do not go out of bounds, you can specify the x and y size limits in the function call. Some threads may be idle, but you still get performance benefits.

Striding:
When you don't have enough threads or want to run on fewer threads.
Use a for loop to move in multiples of a stride length across your data.
This partitions your data so that multiple data elements are being processed per thread.
The stride length is the total number of threads you have executing in the kernel.
With the for loop, you still maintain the bounds check, ensuring you don't reach out of bounds on some elements.

Grid and Block Dimensions:
The number of blocks and threads per block can be specified with 3-vectors.
This is useful for algorithms with 2D and 3D layouts.
Hence, there are gridDim.x, gridDim.y, and gridDim.z.
You can specify the number of threads and blocks per thread with the "threads per block" parameter.
When you launch a kernel, it is on a grid. Each element in the grid is a block, limited to 1024 threads.

You can use .x and .y to access each block dimension within the actual kernel.

Matrix Multiply Example:
The standard form is a triple for-loop form.
How to write/parallelize using CUDA:

One could assign each for-loop an element of the output array and instruct the thread to compute that dot product value.
Each element Cij can be computed in parallel. Each thread corresponds to some element of the output array C. That thread will be responsible for computing the dot product of the corresponding row and column of input matrices A and B.
It can be made to compute the corresponding row and column of a 3D grid.
Block size is a performance parameter of CUDA code.
The code will launch M x N threads.
Thread ij computes Cij.

Issues with it:
Performance issue: Accessing arrays input matrices A and B in an inefficient way.
Each thread will have to load the entirety of row A and B from global memory.
Each thread reads in once, globally it happens m times.
A is read N times.
B is read M times.
Poor data re-use: Reading matrices too many times, every value of A & B is loaded from global memory.
L2 cache and global memory are significantly slower to read from than L1 cache.

How to improve data re-use?
Matrix Multiply with Shared Memory:
Block the computation, specifically the memory accesses.
Divide it up into subblocks. Save each subblock into shared memory as we go. This allows for faster reads as we move into each subblock.
Synchronizing will cause all the threads to wait. All threads will hit a barrier and then continue when all have hit the barrier.

Block (i, j) computes Cij submatrix.
Save A & B submatrices into shared memory.
Accumulate partial dot product into C.
Copy memory into the block.
Synchronize.
Every thread does partial accumulation.
Synchronize.
Move to the next block, going further down the row and down the column.

This reduces the number of reads from A by block-sizing them.
Reduces the number of reads from B by block-sizing it.
We've only done N / block_size times read of A from global memory.
M / block_size times read of B from global memory.
Data reads from global memory are reduced by an order of the block size.
Shared memory

shared keyword denotes shared memory.
syncthreads synchronizes all threads in the block.

Reversing with Shared Memory:
Using shared memory to reverse an array.
Allocate N int in block.
Store into shared memory. Synchronize. Load from shared memory.

The optimization results in a significant speedup:
From 1.997 seconds in a simple GPU program where each ij does the complete dot product.
To 0.091 seconds with optimization in the test.
This demonstrates a significant time savings with optimization. Such examples motivate these memory optimizations on the GPU.

Profiling GPUs:
HPCToolkit + Hatchet:
Additional flags like -e gpu=nvidia will record many additional metrics.
Use hpcstruct <measurements_dir> for structuring the measurements.
NSight:
Use the nsys command to profile.
NVIDIA's own suite of tools for profiling with GPUs.

Roofline Plot:
X-axis: Computation per memory loaded in.
Y-axis: Computation per second.
When you plot this, you end up with a graph that resembles a roofline. It shows the region where you are memory-bound and where you are compute-bound.
Being memory-bound means the processor is idle while waiting for memory to load. This can indicate the need for memory optimizations.

Streams:
Every kernel within in a stream will launch in order. Things launched in same stream start sequentially.
Kernels launched in different streams can start at same time.

Streams:
Every kernel within a stream will launch in order. Kernels launched in the same stream start sequentially.
Kernels launched in different streams can start at the same time.

Unified Memory:
You just create one pointer, and the system manages data for you, instead of having two copies of every pointer, one on the CPU and one on the GPU.
CUDA Managed Memory: The runtime system and hardware will ensure that the pointer is synchronized. This incurs a small overhead.

Higher-Level GPU Programming:

Libraries exist for:

-Linear Algebra
-Signal Processing
-Deep Learning
-Graphics
-Algorithms and Data Structures
These libraries have optimized CUDA code for specific topics.

Big Picture:
When to use GPUs?
Use for data parallel tasks & large datasets.
Consider performance/price and time-to-solution.
What software/algorithm to use?
For performance-critical tasks, consider native languages.
For development time and maintainability, consider higher-level APIs.