Data Parallel Computing and CUDA

Mark Greenstreet and Ian M. Mitchell

CpSc 418 – February 9, 2018

- Data Parallel Computing: slide **??**
  - Computation that does the “same” thing to lots of data
  - Such problem are good candidates for parallel computation.
  - Example: training neural networks
- CUDA: Data Parallel Computing on GPUs
  - GPUs and parallelism
  - Program structure: slide 12
  - Memory: slide 14
  - A simple example: slide 15
  - Launching kernels: slide 22

Data Parallelism

- When you see a `for`-loop:
  - Is the loop-index used as an array index?
  - Are the iterations independent?
  - If so, you probably have data-parallel code.
- Data-Parallel problems:
  - Run well on GPUs because each element (or segment) of the array can be handled by a different thread.
  - Data parallel problems are good candidate for most parallel techniques because the available parallelism grows with the problem size.
  - Compare with “task parallelism” where the problem is divided into the same number of tasks regardless of its size.

Which of the following loops are data parallel?

```
for(int i = 0; i < N; i++)
  c[i] = a[i] + b[i].
```

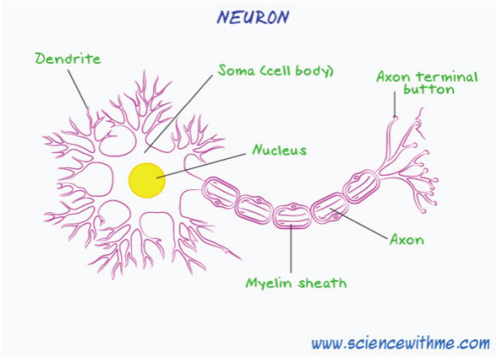
```
dotprod = 0.0;
for(int i = 0; i < N; i++)
  dotprod += a[i]*b[i];
```

```
for(int i = 1; i < N; i++)
  a[i] = 0.5*(a[i-1] + a[i]);
```

```
for(int i = 1; i < N; i++)
  a[i] = sqrt(a[i-1] + a[i]);
```

```
for(int i = 0; i < M; i++) {
  for(int j = 0; j < N; j++) {
    sum = 0.0;
    for(int k = 0; k < L; k++)
      sum += a[i,k]*b[k,j];
    c[i,j] = sum;
} }
```

Neurons



- Dendrites are inputs to other neurons.
- Axon terminals are output to other neurons.
- Simple model:
  - a neuron computes a weighted sum of its inputs – each input is 0 or 1.
  - if this sum is greater than a threshold, the neuron "fires" – it's output becomes one.

$$\mathit{Output} = \begin{cases} 1, & \text{if } \sum_i \mathit{Weight}_i \mathit{Input}_i > \mathit{Threshold} \\ 0, & \text{otherwise} \end{cases}$$

- We can revise the model to make it
  - More biologically accurate – this is what neuro-biologists do.
  - Easier to evaluate on a computer – this is what machine learning people do.

Neural Networks, one layer

- Lots of inputs, and lots of outputs.
- A single input can influence many outputs:
  - Real neurons can have thousands of connections.
  - Three-dimensional wiring (in the brain) allows for complicated interconnection.
- For machine learning:

$$Output = Threshold(W\ Input)$$

  - *Input* is a **vector** of input values.
  - *W* is a matrix of weights. Each row of the matrix models a neuron.
  - *Threshold* is a function that is applied to each element of $W * Input$.
  - To keep the computation tractable:
    - ★ *W* has many inputs and many outputs but is linear.
    - ★ *Threshold* has one input and one output but is non-linear.
    - ★ We don't try to handle multi-in, multi-out, and non-linear all at the same time.

Deep Neural Networks

$$
\begin{aligned}
\mathit{Mid}_1 &= \mathit{Threshold}_1\,(W_1\,\mathit{Input}) \\
\mathit{Mid}_2 &= \mathit{Threshold}_2\,(W_2\,\mathit{Mid}_1) \\
\mathit{Output} &= \mathit{Threshold}_N\,(W_N\,\mathit{Mid}_{N-1})
\end{aligned}
$$

- The first layer of neurons computes the inputs to the next layer.
- We keep going until we get to the output.
- In theory, two-layers can compute anything.
  - Deeper networks can be much smaller in total size – this is what made Geoff Hinton famous.
  - But, deep networks are hard to train.

Training Neural Networks (Supervised)

- Bring lots of dog treats. ☺
- Let's say that I want to train a network to recognize all goats in a photograph.
- Find millions of photographs with the goats (if they have any) labeled.
- Set up a neural network with random values for the elements of the W matrices.
- Calculate the error metric of the network (initially, $Error \geq Awful$)
- Calculate the derivative of the error with respect to the elements of the matrices.
- Adjust the coefficients to lower error.
- Repeat a whole lot of times.
- End product: a neural network that can recognize goats in pictures as well as an expert goat herd.

Why we care (in CpSc 418)

- Training neural networks is very data parallel.
  - E.g., we can calculate the errors from each photo, and them combine them with reduce.
  - If we've got millions of photos, nearly all of the time is spent computing the gradients (i.e. the derivatives).
  - We can process each photo independently – **in parallel**
- Note that we'll be doing lots of matrix-vector and matrix-matrix multiplications.
  - This is officially a machine learning class; so we won't be looking for goat in photos here.
  - But, we will see that many GPU/CUDA applications emphasize algorithms such as matrix multiplication.
  - Machine learning is a big motivation behind the huge grown in popularity of matrix multiplication.

GPUs and Data Parallelism

- GPUs designed for data-parallel computing
  - Each polygon or pixel can be an independent parallel computation.
- GPUs designed for numerical computation
  - Shading, coordinate transformations, physical animation are all numerical computation problem.s
- GPUs have become more programmable to handle a wider range of graphics tasks.
  - In the past 10-15 years, GPUs have become programmable enough that they are useful for scientific computing and machine learning.
  - At first, this was done by hard-core graphics/scientfic computing people who figured out how to implement scientific computing libraries using OpenGL!
  - nVidia saw an opportunity and created CUDA to make it easier.
    - ★ OpenCL is a vendor independent alternative to CUDA.
    - ★ We use CUDA because presently it has more comprehensive support and is easier for getting started.

Key Features of GPU Architectures

- GPUs are Single-Instruction, Multiple-Data (SIMD) machines
  - Each instruction is executed for many data streams using many pipelines.
  - This amortizes the cost of instruction fetch, decode, and control.
  - The lock-step execution of the pipelines simplifies synchronization issues.
- GPUs have deep pipelines
  - Breaking instruction execution into small steps allows simple hardware to get good performance.
  - No bypasses – each instruction must go all the way through the pipeline before another instruction can use the results.
- GPUs have many execution units
  - Typically 8 to 100+ SIMD processors, where each SIMD processor has 32-128 pipelines.
  - A total of 1000 to 10000 pipelines executing in parallel.
- Memory accesses are a **major** bottleneck
  - With so many pipelines, a high-end GPU can perform $\sim 10^{13}$ floating point operations per second.
  - Memory bandwidth is $\sim 5.5 \cdot 10^{11}$ bytes per second. With 4-bytes per single precision floating point number, we need ~ 70 floating point operations per memory read or write to keep the pipelines busy.

CUDA – the programmers view

Threads, warps, blocks, and CGMA – oh my!

- How does the programmer cope with SIMD?
  - Lots of threads – each thread runs on a separate pipeline.
  - A group of thread that execute together, on on each pipeline of a SIMD core are called “a warp”.
- How does the programmer cope with long pipeline latencies, $\sim 30 \textit{cycles}$?
  - Lots of threads – interleave threads so that **other** threads dispatch instructions while waiting for result of current instruction.
  - Note that the need for threads to use multiple pipelines and the need to use threads to high pipeline latency **are multiplicative**
  - CUDA programs have thousands of threads.
- How does the programmer use many SIMD cores?
  - Multiple blocks of threads.
  - Why are threads partitioned into blocks?
    - ★ Threads in the same block can synchronize and communicate easily – they are running on the same SIMD core.
    - ★ Threads in different blocks cannot communicate with each other.
    - ★ There is some relaxation of this constraint in the latest GPUs.

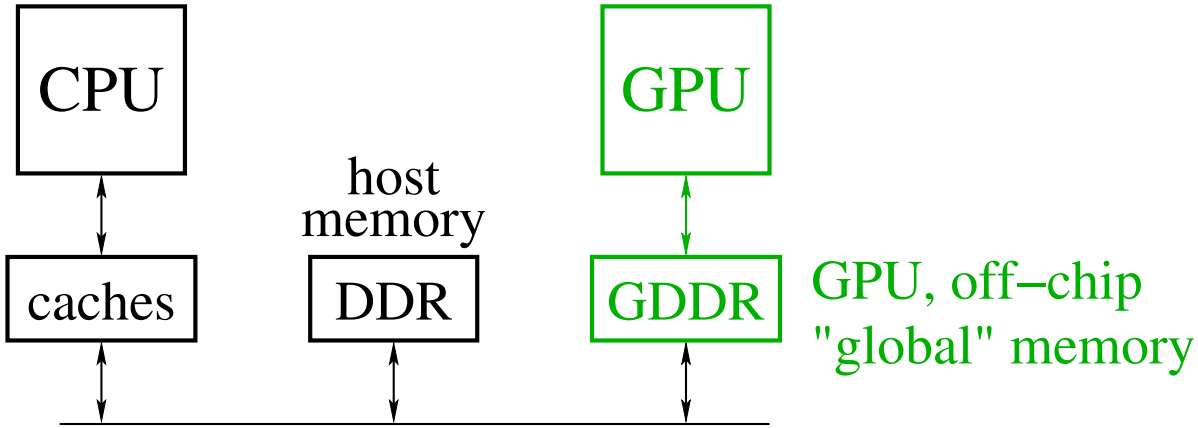
CUDA Program Structure

- A CUDA program consists of three kinds of functions:
  - Host functions:
    - ★ callable from code running on the host, but not the GPU.
    - ★ run on the host CPU;
    - ★ In CUDA C, these look like normal functions – they can be preceded by the `__host__` qualifier.
  - Device functions.
    - ★ callable from code running on the GPU, but not the host.
    - ★ run on the GPU;
    - ★ In CUDA C, these are declared with a `__device__` qualifier.
  - Global functions
    - ★ called by code running on the host CPU,
    - ★ they execute on the GPU.
    - ★ In CUDA C, these are declared with a `__global__` qualifier.

Structure of a simple CUDA program

- A `__global__` function to called by the host program to execute on the GPU.
  - There may be one or more `__device__` functions as well.
- One or more host functions, including `main` to run on the host CPU.
  - Allocate device memory.
  - Copy data from host memory to device memory.
  - "Launch" the device kernel by calling the `__global__` function.
  - Copy the result from device memory to host memory.

Execution Model: Memory



- Host memory: DRAM and the CPU's caches
  - Accessible to host CPU but not to GPU.
- Device memory: GDDR DRAM on the graphics card.
  - Accessible by GPU.
  - The host can initiate transfers between host memory and device memroy.
- The CUDA library includes functions to:
  - Allocate and free device memory.
  - Copy blocks between host and device memory.
  - BUT host code can't read or write the device memory directly.

Example: `saxpy`

`saxpy` = "Scalar `a` times `x` plus `y`".

- The device code.
- The host code.
- The running `saxpy`

saxpy: device code

```
__global__ void saxpy(uint n, float a, float *x, float *y) {
  uint i = blockIdx.x*blockDim.x + threadIdx.x; // nvcc built-ins
  if(i < n)
    y[i] = a*x[i] + y[i];
}
```

- Each thread has `x` and `y` indices.
  - We'll just use `x` for this simple example.
- Note that we are creating one thread per vector element:
  - Exploits GPU hardware support for multithreading.
  - We need to keep in mind that there are a large, but limited number of threads available.

saxpy: host code (part 1 of 5)

```
int main(int argc, char **argv) {
  uint n = atoi(argv[1]);
  float *x, *y, *yy;
  float *dev_x, *dev_y;
  int size = n*sizeof(float);
  x = (float *)malloc(size);
  y = (float *)malloc(size);
  yy = (float *)malloc(size);
  for(int i = 0; i < n; i++) {
    x[i] = i;
    y[i] = i*i;
  }
  ...
}
```

- Declare variables for the arrays on the host and device.
- Allocate and initialize values in the host array.

saxpy: host code (part 2 of 5)

```
int main(void) {
  ...
  cudaMalloc((void**)(&dev_x), size);
  cudaMalloc((void**)(&dev_y), size);
  cudaMemcpy(dev_x, x, size, cudaMemcpyHostToDevice);
  cudaMemcpy(dev_y, y, size, cudaMemcpyHostToDevice);
  ...
}
```

- Allocate arrays on the device.
- Copy data from host to device.

saxpy: host code (part 3 of 5)

```
int main(void) {
  ...
  float a = 3.0;
  saxpy<<<ceil(n/256.0),256>>>(n, a, dev_x, dev_y);
  cudaMemcpy(yy, dev_y, size, cudaMemcpyDeviceToHost);
  ...
}
```

- Invoke the code on the GPU:
  - `add<<<ceil(n/256.0),256>>>(...)` says to create $\lceil n/256 \rceil$ blocks of threads.
  - Each block consists of 256 threads.
  - See slide 23 for an explanation of threads and blocks.
  - The pointers to the arrays (in device memory) and the values of `n` and `a` are passed to the threads.
- Copy the result back to the host.

saxpy: host code (part 4 of 5)

```
  ...
  for(int i = 0; i < n; i++) { // check the result
    if(yy[i] != a*x[i] + y[i]) {
      fprintf(stderr,
              "ERROR: i=%d, a[i]=%f, b[i]=%f, c[i]=%f\n",
              i, a[i], b[i], c[i]);
      exit(-1);
    }
  }
  printf("The results match!\n");
  ...
}
```

- Check the results.

`saxpy`: host code (part 5 of 5)

```
int main(void) {
  ...
  free(x);
  free(y);
  free(yy);
  cudaFree(dev_x);
  cudaFree(dev_y);
  exit(0);
}
```

- Clean up.
- We're done.

Launching Kernels

- Terminology
  - Data parallel code that runs on the GPU is called a **kernel**.
  - Invoking a GPU kernel is called **launching** the kernel.
- How to launch a kernel
  - The host CPUS invokes a `__global__` function.
  - The invocation needs to specify how many threads to create.
  - Example:
    - ★ `add<<<ceil(n/256.0),256>>>(...)`
    - ★ creates $\lceil \frac{n}{256} \rceil$ **blocks**
    - ★ with 256 **threads** each.

Threads and Blocks

- The GPU hardware combines threads into **warps**
  - Warps are an aspect of the hardware.
  - All of the threads of warp execute together – this is the SIMD part.
  - The functionality of a program doesn't depend on the warp details.
  - But understanding warps is critical for getting good performance.
- Each warp has a "next instruction" pending execution.
  - If the dependencies for the next instruction are resolved, it can execute for all threads of the warp.
  - The hardware in each streaming multiprocessor dispatches an instruction each clock cycle if a ready instruction is available.
  - The GPU in `lin25` supports 32 such warps of 32 threads each in a "thread block."
- What if our application needs more threads?
  - Threads are grouped into "thread blocks".
  - Each thread block has up to 1024 threads (the HW limit).
  - The GPU can swap thread-blocks in and out of main memory
    - ★ This is GPU system software that we don't see as user-level programmers.

Compiling and running

```
lin25$ nvcc saxpy.cu -o saxpy
lin25$ ./saxpy 1000
The results match!
```

But is it fast?

- For the `saxpy` example as written here, not really.
  - Execution time dominated by the memory copies.
- But, it shows the main pieces of a CUDA program.
- To get good performance:
  - We need to perform many operations for each value copied between memories.
  - We need to perform many operations in the GPU for each access to global memory.
  - We need enough threads to keep the GPU cores busy.
  - We need to watch out for thread divergence:
    - ★ If different threads execute different paths on an if-then-else,
    - ★ Then the else-threads stall while the then-threads execute, and vice-versa.
  - And many other constraints.
- GPUs are great if your problem matches the architecture.

Preview

February 13: Tuesday – Mark's office hours	
	HW 4 goes out – midterm review, maybe some simple CUDA
February 14: GPU Architecture	
Reading:	Kirk & Hwu – Chapter 3
Homework:	HW 3 earlybird (1:00pm).
PIKAs:	PIKA 4 goes out.
February 15:	
Homework:	HW 3 due (1:00pm).
February 16: Midterm Review	
PIKAs:	PIKA 4 due (1:00pm).
February 19-23: break week	
February 28: midterm – see next slide	

Review

- What is data parallelism?
- What is SIMD execution?
- Think of a modification to the `saxpy` program and try it.
  - You'll probably find you're missing programming features for many things you'd like to try.
  - What do you need?
  - Stay tuned for upcoming lectures.