Introduction to GPGPUs

Mark Greenstreet

CpSc 418 – Mar. 3, 2017

- GPUs
  - Early geometry engines.
  - Adding functionality and programmability.
  - GPGPUs
- CUDA
  - Execution Model
  - Memory Model
  - A simple example

Before the first GPU

Early 1980's: bit-blit hardware for simple 2D graphics.

- Draw lines, simple curves, and text.
- Fill rectangles and triangles.
- Color used a "color map" to save memory:
  - bit-wise logical operations on color map indices!

1989: The SGI Geometry Engine

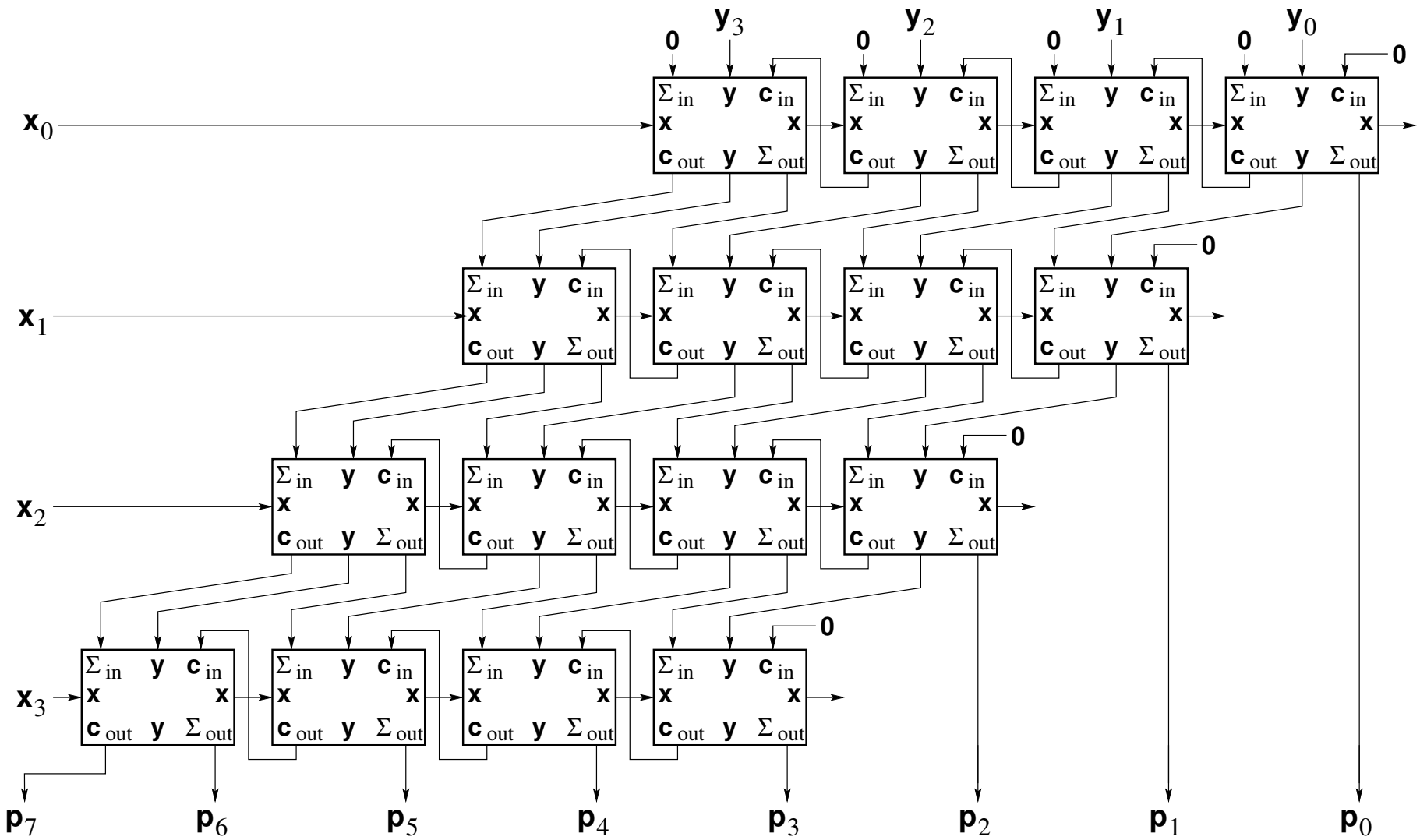
- Basic rendering: coordinate transformation.
  - Represent a 3D point with a 4-element vector.
  - The fourth element is 1, and allows translations.
  - Multiply vector by matrix to perform coordinate transformation.
- Dedicated hardware is much more efficient that a general purpose CPU for matrix-vector multiplication.
  - For example, a 32×32 multiplier can be built with $32^2 = 1024$ one-bit multiplier cells.
    - ★ A one-bit multiplier cell is about 50 transistors.
    - ★ That's about 50K transistors for a very simple design.
      30K is quite feasible using better architectures.

1989: The SGI Geometry Engine

- Basic rendering: coordinate transformation.
- Dedicated hardware is much more efficient that a general purpose CPU for matrix-vector multiplication.
  - For example, a 32×32 multiplier can be built with $32^2 = 1024$ one-bit multiplier cells.
    - ★ A one-bit multiplier cell is about 50 transistors.
    - ★ That's about 50K transistors for a very simple design.
      30K is quite feasible using better architectures.
  - The 80486DX was also born in 1989.
    - ★ The 80486DX was 1.2M transistors, 16MHz, 13MIPs.
    - ★ That's equal to 24 dedicated multipliers.
    - ★ 16 multiply-and-accumulate units running at 50MHz (easy in the same 1μ process) produce 1.6GFlops!

Why is dedicated hardware so much faster?

A simple multiplier



Latency and period are $2N$.

Building a better multiplier

- Simple multiplier has latency and period of $2N$.
- Pipelining: add registers between rows.
  - The period is N, but the latency is N^2.
  - The bottleneck is the time for carries in each row.
- Use carry-lookahead adders (compute carries with a scan)
  - period is $\log N$, the latency is $N \log N$.
  - but the hardware is more complicated.
- Use carry-save adders and one carry-lookahead at the end
  - each adder in the multiplier forwards its carriers to the next adder.
  - the final adder resolves the carries.
  - period is 1, latency is N.
  - and the hardware is **way** simpler than a carry-lookahead design
- Graphics and many scientific and machine learning computations are very tolerant of latency.

Why is dedicated hardware so much faster?

Example: matrix-vector multiplication

- addition and multiplication are “easy”.
- it’s the rest of CPU that’s complicated and the usual performance bottleneck
  - memory read and write
  - instruction fetch, decode, and scheduling
  - pipeline control
  - handling exceptions, hazards, and speculation
  - etc.
- GPU architectures amortize all of this overhead over a lot of execution units.

The fundamental challenge of graphics

Human vision isn't getting any better.

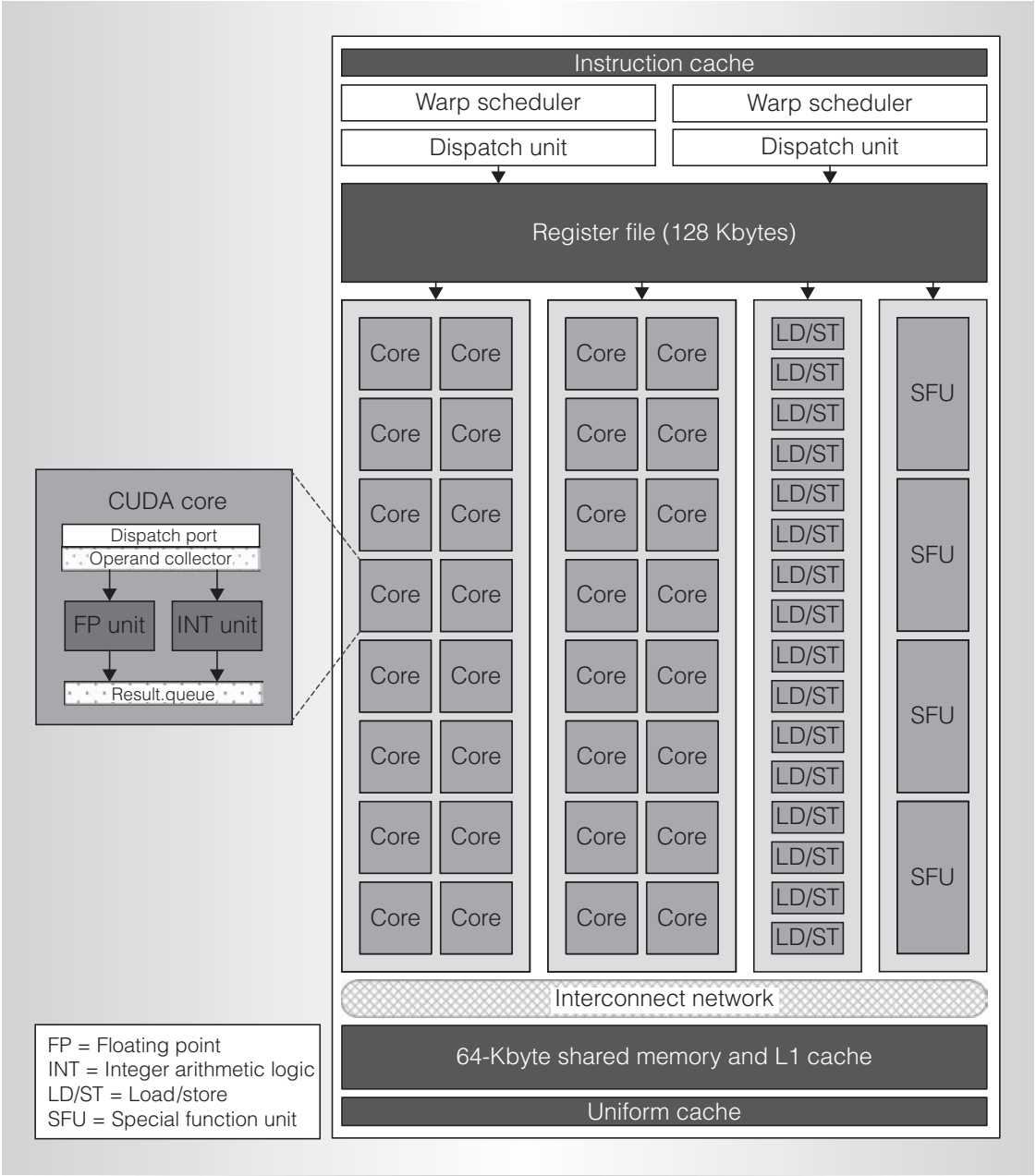
- Once you can perform a graphics task at the limits of human perception (or the limits of consumer budget for monitors), then there's no point in doing it any better.
- Rapid advances in chip technology meant that coordinate transformations (the specialty of the SGI Geometry Engine) were soon as fast as anyone needed.
- Graphics processors have evolved to include more functions. For example,
  - Shading
  - Texture mapping
- This led to a change from hardwired architectures, to programmable ones.

The GPGPU

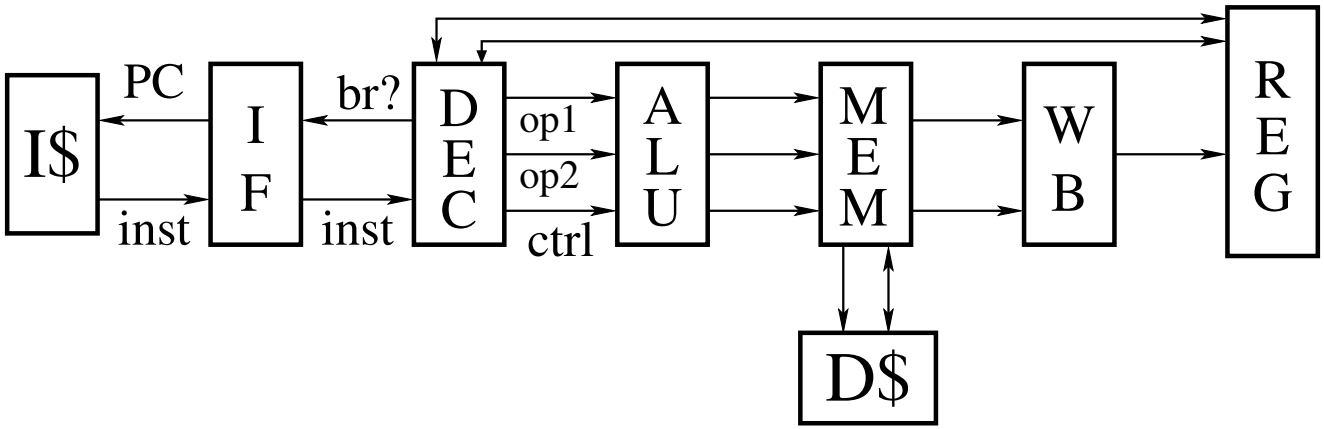
General Purpose Graphics Processing Unit

- The volume market is for graphics, and the highest profit is GPUs for high-end gamers.
  - Most of the computation is floating point.
  - Latency doesn't matter.
  - Abundant parallelism.
- Make the architecture fit the problem:
  - SIMD – single instruction, multiple (parallel) data streams.
    - ★ Amortize control overhead over a large number of functional units.
    - ★ They call it SIMT (. . . , multiple threads) because they allow conditional execution.
  - High-latency operations
    - ★ Allows efficient, high-throughput, high-latency floating point units.
    - ★ Allows high latency accesses to off-chip memory.
  - This means lots of threads per processor.

The Fermi Architecture



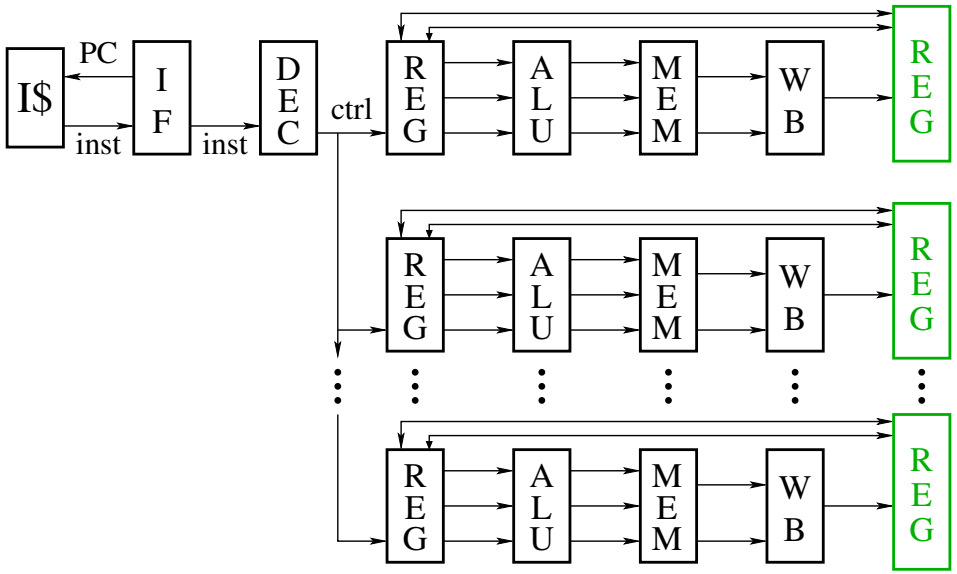
What does a core look like?



A RISC Pipeline

- RISC pipeline: see Jan. 23 slides (e.g. slides 5ff.)
  - Instruction fetch, decode and other control takes much more power than actually performing ALU and other operations!
- SIMD: Single-Instruction, Multiple-Data
- What about memory?

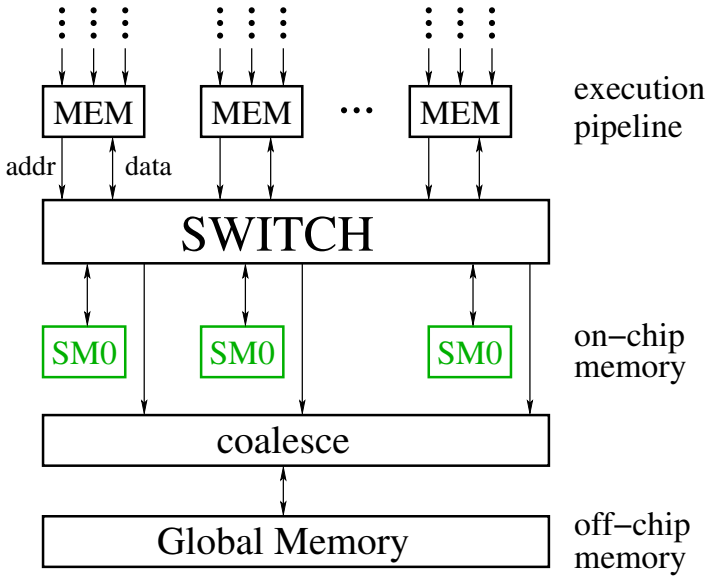
What does a core look like?



A SIMD Pipeline

- RISC pipeline: see Jan. 23 slides (e.g. slides 5ff.)
- SIMD: Single-Instruction, Multiple-Data
  - Multiple execution pipelines execute the same instructions.
  - Each pipeline has its own registers and operates on separate data values.
  - Commonly, pipelines access **adjacent** memory locations.
  - Great for operating on matrices, vectors, and other arrays.
- What about memory?

What does a core look like?



Memory Architecture

- RISC pipeline: see Jan. 23 slides (e.g. slides 5ff.)
- SIMD: Single-Instruction, Multiple-Data
- What about memory?
  - On-chip “shared memory” switched between cores: see Jan. 25 slides (e.g. slide 3)
  - Off-chip references are “coalesced”: the hardware detects reads from (or writes to) consecutive locations and combines them into larger, block transfers.

More about GPU Cores

- Execution pipeline can be very deep – 20-30 stages.
  - Many operations are floating point and take multiple cycles.
  - A floating point unit that is deeply pipelined is easier to design, can provide higher throughput, and use less power than a lower latency design.
- No bypasses
  - Instructions block until instructions that they depend on have completed execution.
  - GPUs rely on extensive multi-threading to get performance.
- Branches use **predicated execution**:
  - Execute the then-branch code, disabling the “else-branch” threads.
  - Execute the else-branch code, disabling the “then-branch” threads.
  - The order of the two branches is unspecified.
- Why?
  - All of these choices optimize the hardware for graphics applications.
  - To get good performance, the programmer needs to understand how the GPGPU executes programs.

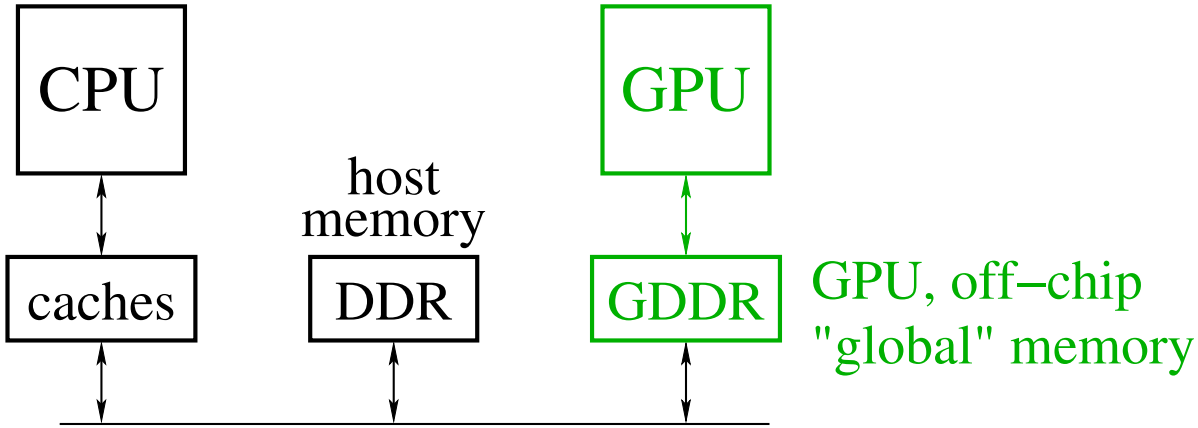
Lecture Outline

- GPUs
  - been there, done that.
- CUDA – we are here!
  - Execution Model
  - Memory Model
  - Code Snippets

Execution Model: Functions

- 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.
  - 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.

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 memory.
- 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.

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.
- We'll do the `saxpy` example from the paper.
  - `saxpy` = "Scalar `a` times `x` plus `y`".

`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 22 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.

Threads and blocks

- Our example created $\left\lceil \frac{\texttt{n}}{256} \right\rceil$ **blocks** with 256 **threads** each.
- The GPU hardware has a pool of running threads.
  - Each thread has a "next instruction" pending execution.
  - If the dependencies for the next instruction are resolved, the "next instruction" can execute.
  - The hardware in each streaming multiprocessor dispatches an instruction each clock cycle if a ready instruction is available.
  - The GPU in `lin25` supports 1024 such threads.
- 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-block in and out of main memory
    - ⋆ This is GPU system software that we don't see as user-level programmers.

But is it fast?

- For this example, 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

March 6: Intro. to CUDA Reading Kirk & Hwu Ch. 2
March 8: CUDA Threads, Part 1 Reading Kirk & Hwu Ch. 3
March 10: CUDA Threads, Part 2
March 13: CUDA Memory Reading Kirk & Hwu Ch. 4
March 15: CUDA Memory: examples
March 17: CUDA Performance Reading Kirk & Hwu Ch. 5
March 20: Matrix multiplication with CUDA, Part 1
March 22: Matrix multiplication with CUDA, Part 2
March 24 – April 3: Other Topics • more parallel algorithms, e.g. dynamic programming? • reasoning about concurrency, e.g. termination detection • other paradigms, e.g. Scala and futures?
April 5: Party: 50th Anniversary of Amdahl's Law

Review

- What is SIMD parallelism?
- How does a CUDA GPU handle branches?
- How does a CUDA GPU handle pipeline hazards?
- What is the difference between “shared memory” and “global memory” in CUDA programming.
- 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.