Data Parallel Computing and CUDA

Mark Greenstreet

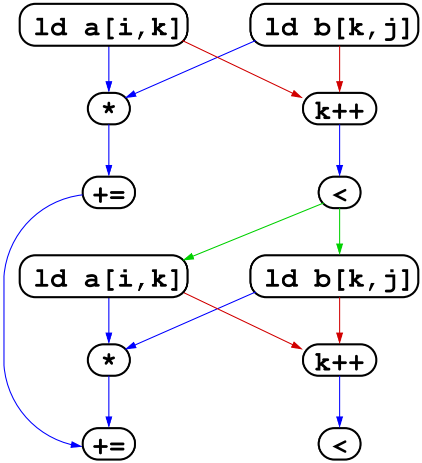
CpSc 418 – October 29, 2018

Outline

- GPU architecture
  - Multiple pipelines
  - No pipeline bypasses
  - Lots of threads
  - Watch out for memory bottlenecks!
- A CUDA example: `saxpy`
  - Program structure: slide 13
  - Memory: slide 15
  - A simple example: slide 16
  - Launching kernels: slide 23

Matrix Multiplication: Dependencies

```
for(k=0; k < L; k++)
  sum += a[i,k]*b[k,j];
```



- Matrix multiplication – innner loop.
- Dependencies:
  - $\rightarrow$: RAW
  - $\rightarrow$: WAR
  - $\rightarrow$: CTRL
  - Not showing dependencies that are dominated by the ones drawn.
    - E.g. not showing WAW from `k++` to `k++` in next iteration.
- Maximum speed up $= 2$
  - Assuming simple architecture: no renaming or branch prediction.
  - This is the GPU approach: simple $\Rightarrow$

    less energy per operation $\Rightarrow$
    higher throughput per GPU.

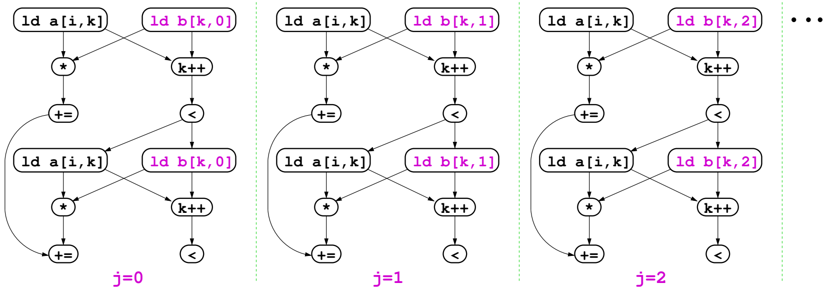
Matrix Multiplication on a RISC Pipeline

```
loop:  ld a[i,k]
       ld b[k,j]
       k++
       *
       b< loop
       += # delay
```

IF	DEC	ALU	MEM	WB	
ld a					
ld b	ld a				
k++	ld b	ld a			
*	k++	ld b	ld a		
b<	*	k++	ld b	ld a	BYPASS: a, b
+=	b<	*	k++	ld b	BYPASS: k
ld a	+=	b<	*	k++	BYPASS: a*b

Data Parallelism

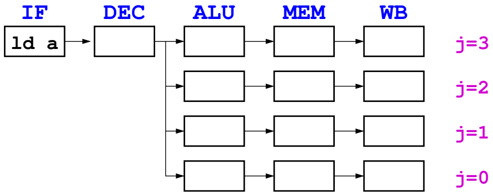
```
for(j=0; j < M; j++)
  for(k=0; k < L; k++)
    sum += a[i,k]*b[k,j];
```



- The iterations of the `j`-loop are independent.
- Maximum speed-up $= 2L$.
- Even more speed-up if we exploit the `i`-loop in the same way.

SIMD Execution

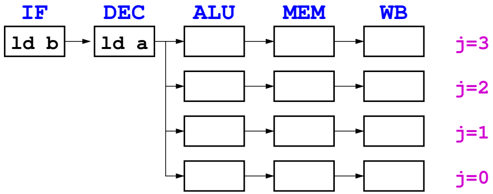
```
loop:   ld a[i,k]
        ld b[k,j]
        k++
        *
        b< loop
        += # delay
```



- SIMD = Single Instruction, Multiple Data
- All execution pipelines (i.e. ALU through WB) share the same IF and DEC stages.

SIMD Execution

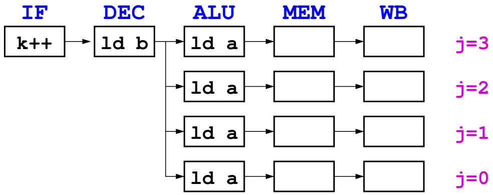
```
loop:  ld a[i,k]
       ld b[k,j]
       k++
       *
       b< loop
       += # delay
```



- SIMD = Single Instruction, Multiple Data
- All execution pipelines (i.e. ALU through WB) share the same IF and DEC stages.

SIMD Execution

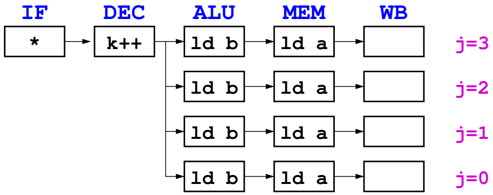
```
loop:   ld a[i,k]
        ld b[k,j]
        k++
        *
        b< loop
        += # delay
```



- SIMD = Single Instruction, Multiple Data
- All execution pipelines (i.e. ALU through WB) share the same IF and DEC stages.

SIMD Execution

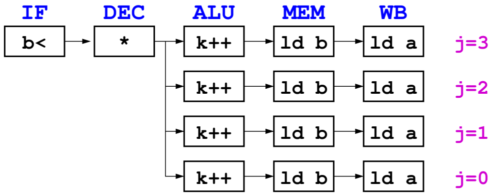
```
loop:  ld a[i,k]
       ld b[k,j]
       k++
       *
       b< loop
       += # delay
```



- SIMD = Single Instruction, Multiple Data
- All execution pipelines (i.e. ALU through WB) share the same IF and DEC stages.

SIMD Execution

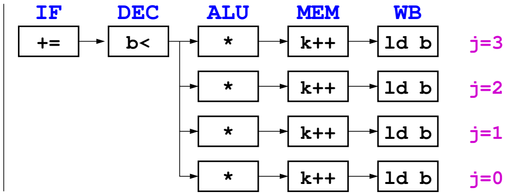
```
loop:  ld a[i,k]
       ld b[k,j]
       k++
       *
       b< loop
       += # delay
```



- SIMD = Single Instruction, Multiple Data
- All execution pipelines (i.e. ALU through WB) share the same IF and DEC stages.
- Bypasses require fast data routing ⇒ bypasses are **energy hogs.**

SIMD Execution

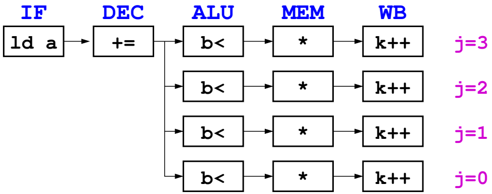
```
loop:  ld a[i,k]
       ld b[k,j]
       k++
       *
       b< loop
       += # delay
```



- SIMD = Single Instruction, Multiple Data
- All execution pipelines (i.e. ALU through WB) share the same IF and DEC stages.
- Bypasses require fast data routing $\Rightarrow$ bypasses are **energy hogs.**

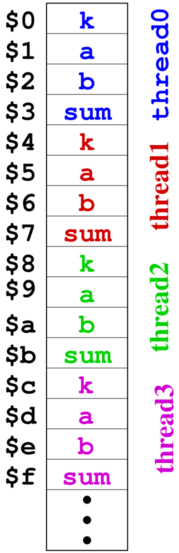
SIMD Execution

```
loop:  ld a[i,k]
       ld b[k,j]
       k++
       *
       b< loop
       += # delay
```



- SIMD = Single Instruction, Multiple Data
- All execution pipelines (i.e. ALU through WB) share the same IF and DEC stages.
- Bypasses require fast data routing ⇒ bypasses are **energy hogs.**

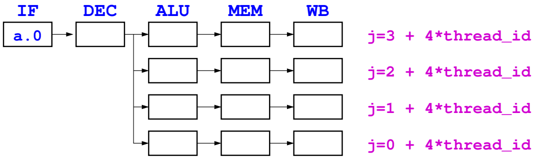
Multithreading: Registers



- Large, physical register file.
- Each thread uses a portion of the registers.
  - E.g. The thread-id can select an offset into the register file.
- Multithreading & CUDA
  - The CUDA compiler determines how many registers each thread needs.
  - This then determines how many threads can execute at the same time.

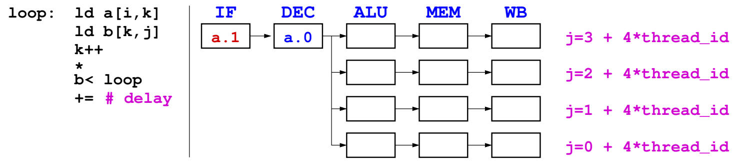
Multithreaded Execution

```
loop:  ld a[i,k]
       ld b[k,j]
       k++
       *
       b< loop
       += # delay
```



- No need for bypasses – just use more threads.
- No need for delay slots or branch prediction – just use more threads.

Multithreaded Execution

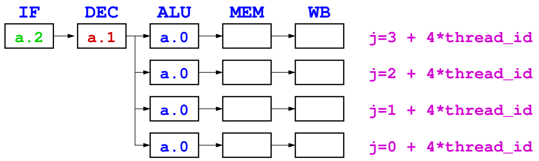


```
loop:  ld a[i,k]
       ld b[k,j]
       k++
       *
       b< loop
       += # delay
```

- No need for bypasses – just use more threads.
- No need for delay slots or branch prediction – just use more threads.

Multithreaded Execution

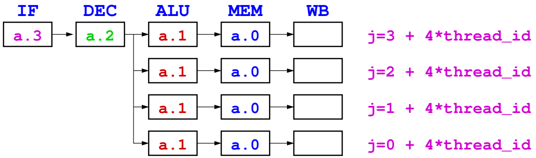
```
loop:  ld a[i,k]
       ld b[k,j]
       k++
       *
       b< loop
       += # delay
```



- No need for bypasses – just use more threads.
- No need for delay slots or branch prediction – just use more threads.

Multithreaded Execution

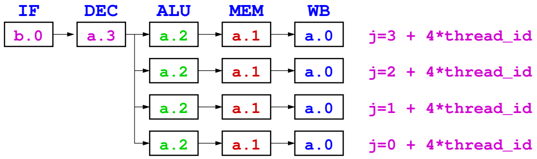
```
loop:  ld a[i,k]
       ld b[k,j]
       k++
       *
       b< loop
       += # delay
```



- No need for bypasses – just use more threads.
- No need for delay slots or branch prediction – just use more threads.

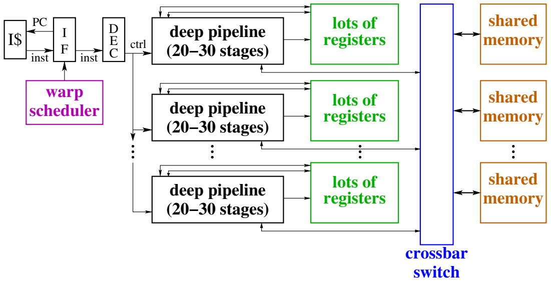
Multithreaded Execution

```
loop:  ld a[i,k]
       ld b[k,j]
       k++
       *
       b< loop
       += # delay
```



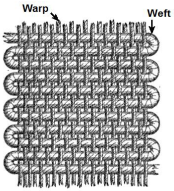
- No need for bypasses – just use more threads.
- No need for delay slots or branch prediction – just use more threads.

A GPU “Streaming Multiprocessor”



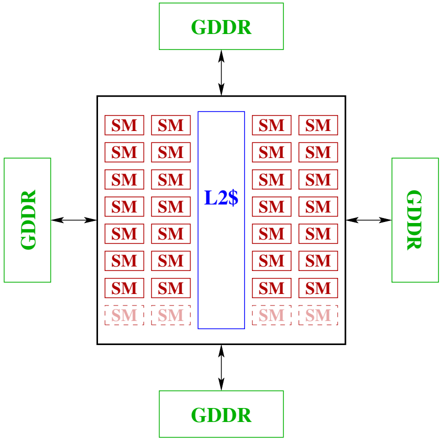
- No pipeline bypasses
  - the pipelines are multi-threaded
  - A group of threads that execute one-per-pipeline is called a “warp”.
  - The warp-scheduler determines which instruction to dispatch next.
- Caches, branches, and other issues
  - We’ll handle them in upcoming lectures.
  - It all comes back to keep the architecture simple and use more threads.

What’s a Warp?



- Warp is a term from weaving – a warp is an arrangment of threads.
- Figure from http://commons.wikimedia.org/wiki/User:Ryj

And there's more



- Each SM is a SIMD pipeline as shown on the previous slide.
  - SIMD = Single-Instruction Multiple-Data
- A latest-and-greatest GPU today has
  - 72 SMs
  - 64 pipelines per SM
  - 12 GDDR memory

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, $\sim 30 cycles$?
  - 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.

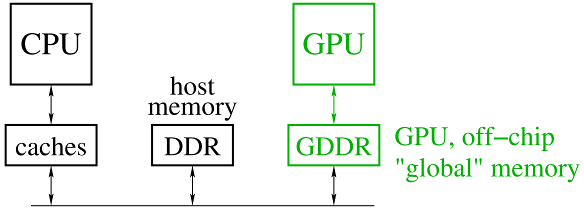
CUDA Program Structure

- 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 24 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 $\left\lceil \frac{n}{256} \right\rceil$ **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

October 31: GPU Threads (part 1)
Reading: Kirk & Hwu – Chapter 3

November 2: GPU Threads (part 2)

November 5: Systolic Algorithms (part 1)

November 7: Systolic Algorithms (part 2)

November 9: GPU Memory
Reading: Kirk & Hwu – Chapter 4

- Kirk & Hwu $=$ *Programming Massively Parallel Computers*
- On-line here – where "here" means `https://www.sciencedirect.com/book/9780128119860/programming-massively-parallel-processors`
- Free access from UBC (use the UBC library ezproxy from off-campus).

Review

- What is SIMD execution?
- What don't GPUs have pipeline bypasses?
- Why do CUDA programs use so many threads?
- 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.