
# ECSE 420
# GPU Crash Course

TA: Loren Lugosch

November 2016

# Outline

- Problems with CPUs
- GPUs to the rescue!
- CUDA programming model
- GPU hardware architecture

# Problems with CPUs

- Review: modern CPUs are very parallel
- Multiple cores (~8)
- Multiple threads per core (~2)
- Multiple operands for SIMD instructions (~4)

  => 64-way parallelism

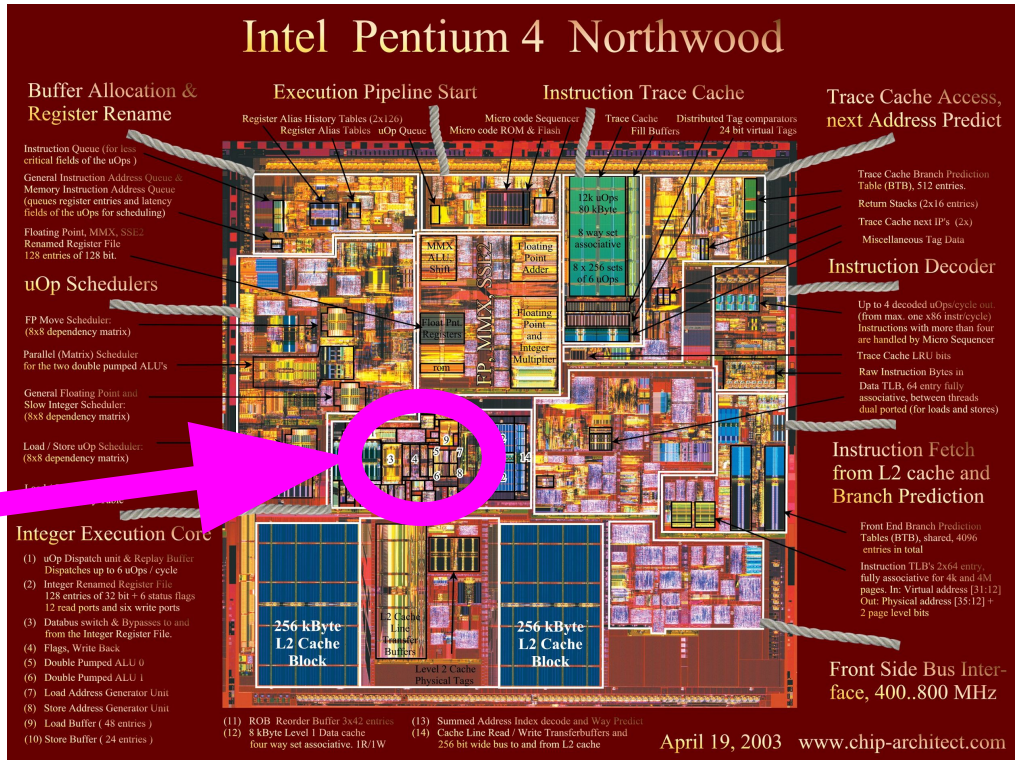
# Problems with CPUs

- But CPUs are not parallel enough for many problems
- Example: rectify
  - P = 1 ⟶ embarrassingly parallel
  - Theoretically, you could rectify a 1 megapixel image in a single clock cycle
  - Our CPU can only process 64 pixels per clock cycle
- CPUs have low **throughput**
  - **Throughput:** number of tasks completed per unit time
  - **Latency:** time to complete one particular task

# Problems with CPUs

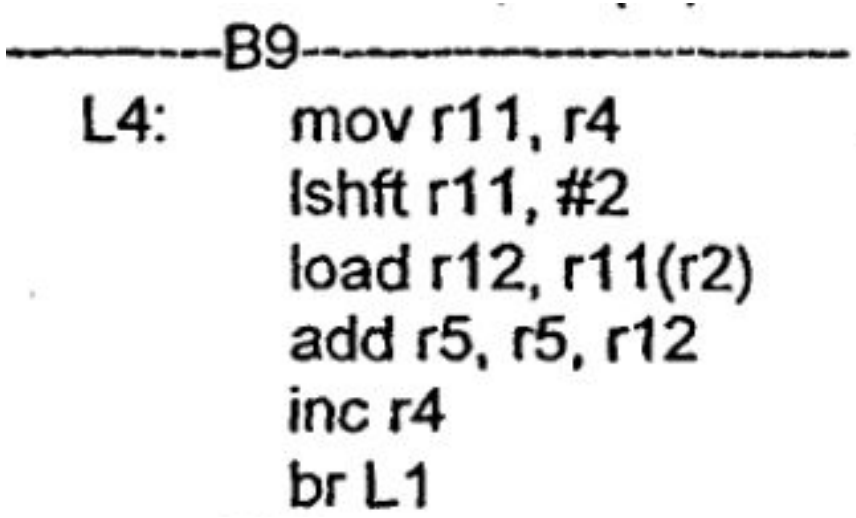
- ALU is the powerhouse of the CPU
- But non-ALU stuff takes up lots of area on CPUs:

ALU



# Problems with CPUs

- Why not have just a bunch of parallel ALUs?
- Because most CPU workload is random, sequential “basic blocks”:



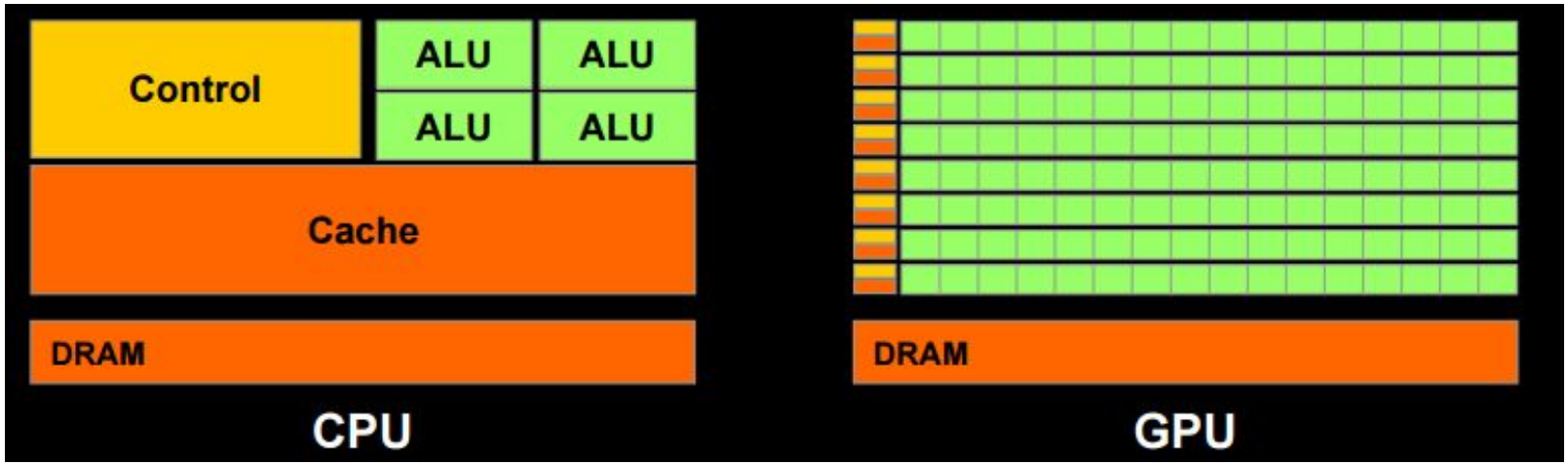

- Non-ALU stuff (pipelining, branch prediction, speculative execution, … ) reduces **latency** for most instructions

# GPUs to the rescue!

- Solution: have latency-optimized CPU + throughput-optimized *<something>*
- People have argued for years about what the *<something>* should be
- For very good reasons, the *<something>* ended up being the **GPU** (Graphics Processing Unit)

# GPUs to the rescue!

- GPUs originally only intended for rendering video
  - Many graphics operations (like rectification) are of the form “Do X to each pixel in parallel”
  - “My computer already has a GPU, and it’s very parallel, so why don’t I just make my GPU the high-throughput *<something>*?”



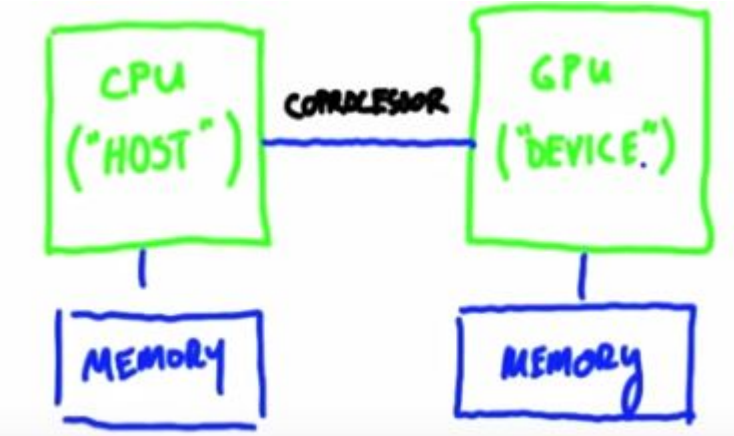
# GPUs to the rescue!

- Some clever programmers hacked GPU graphics libraries to perform general-purpose computing ("GPGPU") instead of graphics
- NVIDIA noticed this trend and released CUDA, a framework for general-purpose computations on the GPU


- There is also OpenCL, which runs on both AMD and NVIDIA (but for Lab 3, we will use CUDA)

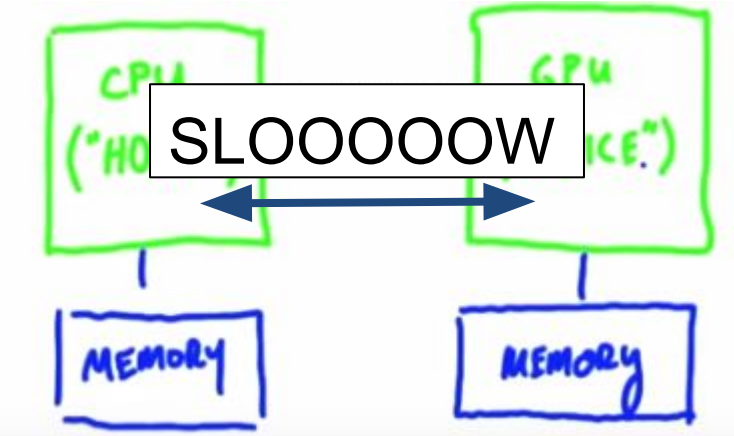
# CUDA programming model

- CUDA has “heterogeneous” programming model
  - Programmer writes one program in a high-level language (CUDA C = C with some extensions)
  - Compiler maps pieces of the code to the “host” (the CPU) and to the “device” (the GPU)
- Host is master
- Host and device have separate memories



# CUDA programming model

- Host is responsible for:
  - moving data from host to device, and device to host
  - allocating memory on the device
  - launching “kernels” on the device
- **Note**: copying memory between CPU and GPU is *very, very slow*-- perform infrequently



# CUDA programming model

- A typical CUDA program looks like this:

1) CPU allocates storage on GPU
2) CPU copies input data from CPU → GPU
3) CPU launches kernel(s) on GPU to process the data
4) CPU copies the results from GPU → CPU

# CUDA programming model

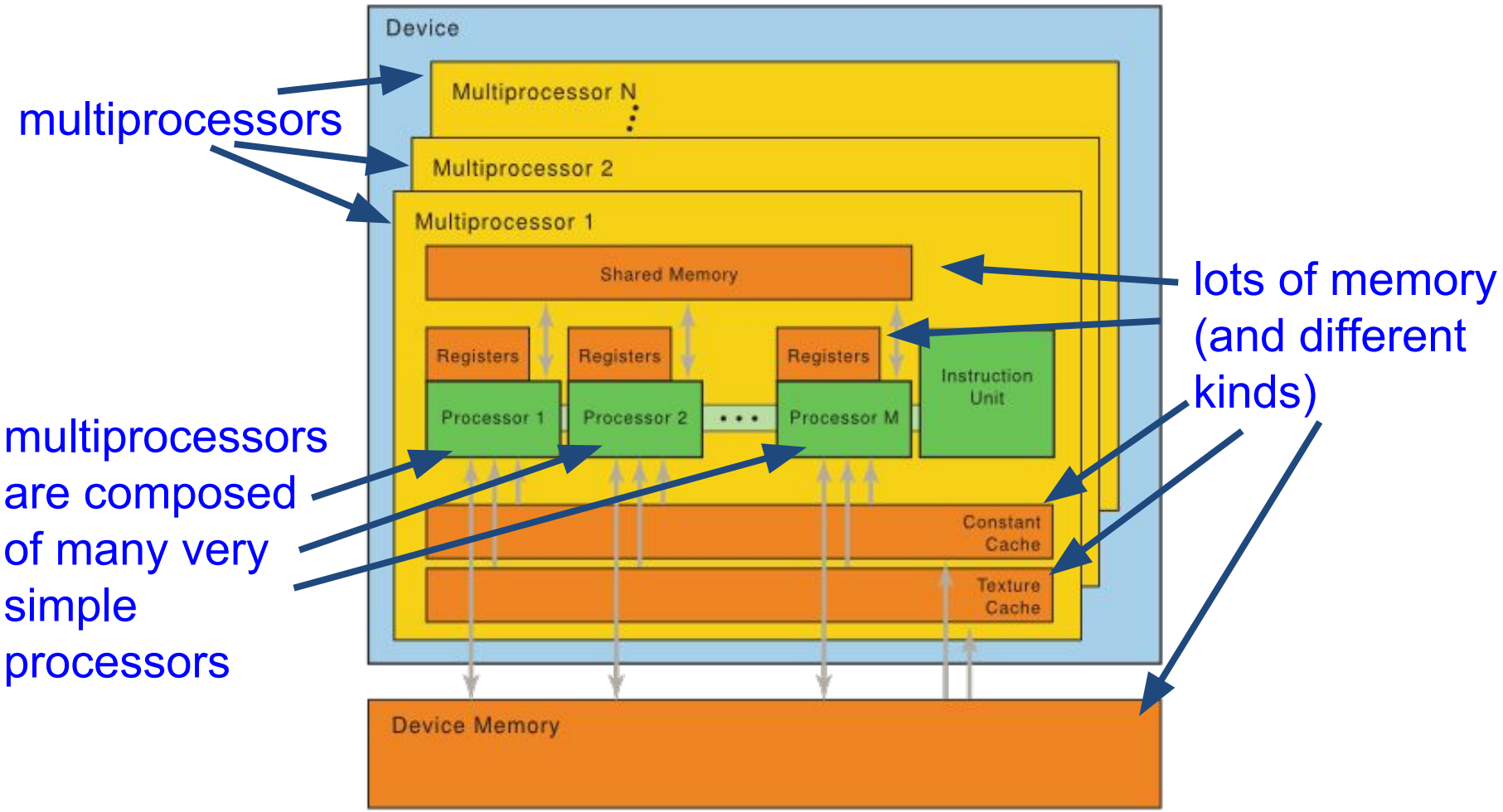
- Once host has launched the kernel, device is responsible for running the kernel
- A kernel looks like a serial program (similar to the function pointer for Pthreads)
- GPU runs this same program on many threads
- Unlike CPU, launching threads on GPU is very efficient
- Don’t be afraid to launch lots and lots of threads (e.g., 1 million!)-- there is very little overhead

# CUDA programming model

- Like OpenMP, CUDA has a few simple techniques for synchronization:
  - barriers
  - atomic operations
- There is a performance penalty for synchronization, but not as bad as on CPU

# GPU hardware architecture

- GPUs look like this:



# GPU hardware architecture

- Threads run in parallel in collections called blocks
- Each block runs on a separate multiprocessor
- The GPU schedules blocks on multiprocessors (not the programmer)
- Suppose GPU has 10 multiprocessors, and programmer uses 20 blocks
- The GPU will:
  - run the 10 blocks in parallel
  - wait until they are finished
  - then run the remaining 10 blocks

# GPU hardware architecture

- Another architectural detail to remember: **SIMD units**
- Typical CPU has instruction decoder and ALU
- Suppose that each processor in a GPU is performing the same operation
  - Then no need for multiple instruction decoders--just have one and connect to many ALUs
- Such a collection is called a SIMD unit
  - Multiple threads executing an instruction using a SIMD unit = **warp**
  - Warps have maximum size of 32 threads (for now!)

# GPU hardware architecture

- Threads doing different things = **thread divergence**
- Why is it good to avoid thread divergence?
- If threads are doing very different things, it is difficult for compiler and instruction scheduler to take advantage of SIMD units
- So avoid using `if()` in kernel code (even if it means threads do redundant work)

# Final note

- You can use Prof. Zilic's GPU for this lab
- SSH into the following address:
  - labgroup<your group number>@132.206.77.19
  - (Hostname: queens1)
- Your code must compile and run on this machine

# References

- Material taken from the Udacity online course “Intro to Parallel Programming” and various NVIDIA pages