Class Information

• **Instructor:** Zheng (Eddy) Zhang  
  Email: eddy.zhengzhang@gmail.com  
  Office: CoRE 310  
  Office Hour: 1pm — 2pm, Wednesdays

• **Lectures (including 2-3 guest lectures from industry)**  
  Time: Tuesday Noon — 3pm  
  Location: TIL-253 LIV (tentative)  
  Course Webpage: Coming Soon

• **Grade Basis**  
  In-class presentation: 10%  
  Three projects: 60%  
  Five homework assignments: 30%
Topics Covered in Class

• **Compiler Construction**
  - Lexical and Syntax Analysis
  - Data Scopes and Intermediate Representation
  - Instruction Scheduling and Resource Scheduling

• **Program Behavior Analysis**
  - Data Flow Analysis
  - Control Flow Analysis
  - Memory and Cache Locality Optimization
  - Correctness, Safety and Liveness

• **Parallelization**
  - Task Decomposition
  - Task Scheduling
  - Parallel Algorithm Primitives
  - Static and Dynamic Performance Tuning
What is a programming language?

- **Languages from the user's point of view**
  - A way of thinking
  - A way of expressing algorithms

- **Languages from the implementor's point of view**
  - An abstraction of virtual machine
  - A way of specifying what you want the hardware to do without getting down into the bits
What is a compiler?

It is a program that translates a program in one language into a program in another language.

"It was our belief that if FORTRAN, during its first months, were to translate any reasonable “scientific” source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger... I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed."

— John Backus
Three Pass Compiler

- **Front End**
  Maps legal source code to intermediate representation (IR). Allows multiple front-ends. Typically front-end problem is $O(n)$ or $O(n \times \log(n))$

- **Middle End**
  Performs language/architecture independent analysis and optimization. Maps from one IR to one IR or multiple IRs. Allows multiple passes

- **Back End**
  Mapping IR into target machine machine code. Allows multiple back-ends. Backend problems are typically NP complete, however practical solutions exist.
Phases of Compilation

Compilation versus Interpretation

• **Pure Compilation**
  The compiler translates the high-level source program into an equivalent target program (typically in machine language), and then goes away:

• **Pure Interpretation**
  Interpreter stays around for the execution of the program
  Interpreter is the locus of control during execution

• **Compilation vs. interpretation**
  Not opposites
  Not a clear-cut distinction
What You Will Learn From This Class

In addition to Fundamental Topics in Program Compilation and Analysis

- How to Write Efficient Parallel Programs
- How to Perform Performance Debugging
- How to Read, Review, and Present Research Papers

This offering of CS 515 has an emphasis on the principles and practice on parallel programming
What You Will Learn From This Class

In addition to Fundamental Topics in Program Compilation and Analysis

• How to Write Efficient Programs
• How to Perform Performance Debugging
• How to Read, Review, and Present Research Papers

You are encouraged to combine your own research interests/ideas/projects with parallel programming techniques learned in this class
Why parallel programming?

- **Frequency Scaling Wall**
  - Frequency scaling has stopped since 2004

- **Power Wall**
  - Difficult to maintain the same power density as transistors get smaller

- **ILP Wall**
  - Instruction level parallelism exploited a lot
  - Sophisticated multi-stage pipeline leads to complexity in design

Source: cpudb.stanford.edu
Why parallel programming?

2004: *Santa Clara, we have a problem!*

More pipeline stages, less efficient, more power.

Just can’t remove > 100 watts without great expense on a desktop.

*All* computing is now **Low Power Computing**!

And more dark silicon papers:


…….

“Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse”, Michael Tayler, UCSD,
Dennard's Scaling & Moore’s Law

“As transistors get smaller, the power density can be kept constant.”
— Robert Dennard

Example: if there is a reduction in a transistor’s linear size by 2, the power it uses can fall by 8 (with voltage and current both halving) if frequency remains the same.

\[
\text{Dynamic Power} \propto C \times F \times V^2
\]

\[
\begin{align*}
C &= \text{capacitance} \\
F &= \text{frequency} \\
V &= \text{voltage}
\end{align*}
\]


“The number of transistors in a dense integrated circuit doubles approximately every two years.”
— Gordon Moore

Moore, Gordon E. “Cramming more components onto integrated circuits”. Electronics Magazine. p. 4. (1965)
Why Moore’s Law and Dennard’s Scaling are both coming to an end?

1. Frequency scaling faster than transistor size
2. Voltage stopped scaling in the 80s-90s
3. Power density can’t remain the same
   — Threshold voltage
   — Leakage current

Dynamic Power $\propto C^2 F V^2$

C = capacitance
F = frequency
V = voltage

A projection made in 1999
What to do with more and more transistors?

1. Multi-core scaling
2. Thread level parallelism
3. Cache and memory level parallelism

Parallel Computing is Ubiquitous

Miniature Devices to Supercomputers

GPU
Supercomputers, desktops, mobile devices, self driving cars.

Raspberry PI
Basic computing science education in developing countries, home automation,

Intel Edison
Wearable device, Internet of Things (IOT), and etc.
Challenges in Parallel Computing

- Writing a parallel program is not easy
  - Task decomposition
  - Dependence analysis
  - Load balancing
  - Resource allocation
  - …

- Not to mention write an efficient parallel program
  - Interactions with algorithm
  - Interaction with software/hardware interface
  - Interaction with microarchitecture

- Language and programming system support
  - Heavily optimized for sequential programs or limited-parallelism programs in the past few decades
  - Exploration for the best parallel programming and execution model is an on-going research topic.

Challenges in Parallel Computing

- Writing a parallel program is not easy
  - Task decomposition
  - Dependence analysis
  - Load balancing
  - Resource allocation
  - …

- Not to mention write an efficient parallel program
  - Interactions with algorithm
  - Interaction with software/hardware interface
  - Interaction with microarchitecture

- Language and programming system support
  - Heavily optimized for sequential programs or limited-parallelism programs in the past few decades
  - Exploration for the best parallel programming and execution model is an on-going research topic.

GPU Computing
GPU Programming

- GPUs
  - A good example of multi-core/many-core scaling
  - Widely used to accelerate large-scale applications: machine learning, scientific simulation, weather prediction, computer vision, bioinformatics, and etc.
  - We will use GPUs as our major programming platform to learn how to compile, develop, and optimize parallel programs. All three projects will be implemented using GPU programming language(s).
GPU Throughput V.S. CPU Throughput

Source: cuda programming guide
https://docs.nvidia.com/cuda/cuda-c-programming-guide/
GPU V.S. CPU Memory Throughput

Source: cuda programming guide
https://docs.nvidia.com/cuda/cuda-c-programming-guide/
GPU Programming Languages/Models

- **Compute Unified Device Architecture (CUDA) - NVIDIA**
- Open Computing Language (OpenCL)
- C++ AMP - from Microsoft
- BrookGPU
- OpenGL / DirectX
- Stream (Brook+) - AMD
- ......
GPU Programming Languages/Models

- Compute Unified Device Architecture (CUDA) - NVIDIA
- Open Computing Language (OpenCL)
- C++ AMP - from Microsoft
- BrookGPU
- OpenGL / DirectX

Andrei Karpathy @karpathy · Sep 3
I’ve only picked up GPU/CUDA at random. Actually working through CUDA book from Chapter1 proving to be something I should have done long ago

25 38 388
Let’s look at a sequential program in C first:

Add vector A and vector B to vector C.

```c
void add_vector (float *A, float *B, float *C, int N){
    for (index = 0; index < N; index++ )
}

void main ( ) { ...
    add_vector (A, B, C, N);
    ...}
```
• Now let’s look at the parallel implementation in CUDA:

Add vector A and vector B to vector C.

```cpp
__global__ void add_vector (float *A, float *B, float *C, int N){
    if (tid < N)
}

void main ( ) { ...
    add_vector <<<<dimGrid, dimBlock>>>> (A, B, C, N);
    ...
}
```

/ilab/users/zz124/cs515_2017/samples/0_Simple/vectorAdd
What is the difference?

**CPU Program**

Add vector A and vector B to vector C.

```c
void add_vector (float *A, float *B, float *C, int N){
    ...
    for (index = 0; index < N; index++)
}

void main ( ) { ...
    add_vector (A, B, C, N);
    ...
}
```

**GPU Program**

Add vector A and vector B to vector C.

```c
__global__ void add_vector (float *A, float *B, float *C, int N){
    ...
    if (tid < N)
}

void main ( ) { ...
    add_vector <<<dimGrid, dimBlock>>> (A, B, C, N);
    ...
}
```
Thread View

- A kernel is executed as a grid of thread blocks.
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution.
  - Efficiently sharing data through a low-latency shared memory (scratch-pad memory).
  - Two threads from two different blocks cannot cooperate by shared memory.
- **Concurrent kernel execution possible**
  - Multiple kernels running simultaneously.
  - Computation can also overlap memory transfer.

Threads in a block are organized into warps (32 threads/warp), and a warp is a SIMD thread group.
Single Instruction Multiple Data (SIMD)

- \( C[\text{index}] = A[\text{index}] + B[\text{index}] \);

  \[
  \begin{align*}
  \text{thread 0: } & \quad C[0] = A[0] + B[0]; \\
  \text{...} \\
  \end{align*}
  \]

  Execute at the same time.

- **Flynn’s taxonomy for modern processors**
  - Single Instruction Single Data (SISD)
  - Single Instruction Multiple Data (MIMD)
  - Multiple Instruction Single Data (MISD)
  - Multiple Instruction Multiple Data (MIMD)

GPU — SIMT
(Single Instruction Multiple Thread)

Simultaneous execution of many SIMD thread groups
An Example of GPU Architecture

• **Nvidia GeForce 8800 GTX**
  - 128 streaming processors (SP) each at 575 MHz
  - Each streaming multiprocessor has 16 KB shared memory.
Thread Life Cycle in Hardware

- Thread blocks are distributed to all the SMs
  - Typically more than 1 block per SM

- One Streaming Multi-processor schedules and executes warps that are ready

- As a thread block completes, resources are freed
  - More thread blocks will be distributed to SMs
Thread Blocks on a SM

• **Threads are assigned to SMs in Block granularity**
  - The number of blocks per SM limited by resource constraints
  - SM in C2075 can host 1536 threads:
    - 256 (threads/block) * 6 blocks or
    - 128 (threads/block) * 12 blocks, and etc.

• **Threads run concurrently**
• **SM assigns/maintains thread id #s**
• **SM manages/schedules thread execution**
Thread Scheduling and Execution

- Warps are scheduling units within a SM
  - If 3 blocks are assigned to an SM and each block has 256 threads, how many warps are there?

- At any point in time, only a limited number of those warps will be selected for execution on one SM
SM Warp Scheduling

• SM hardware implements “zero-overhead” Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible warps are selected for execution on a prioritized scheduling (round-robin/age)
  - All threads in a warp execute the same instruction when selected

• Typically 4 clock cycles needed to dispatch the same instruction for all threads in a warp
SM Warp Scheduling

• SM hardware implements “zero-overhead” Warp scheduling

**Question:**

Assume one global memory access is needed for every 4 instructions, at one time only one instruction can be issued, and it takes 4 cycles to dispatch an instruction for one warp.

*What is the minimum number of warps are needed to tolerate the 320-cycle global memory latency?*
Question:
Assume one global memory access is needed for every 4 instructions, at one time only one instruction can be issued, and it takes 4 cycles to dispatch an instruction for one warp.

What is the minimum number of warps are needed to tolerate the 320-cycle global memory latency?
GPU Computing Resources at ilab

• 1 multi-GPU Machine: "atlas.cs.rutgers.edu"
  - 1x Telsa K40: 12 GB memory, 2880 CUDA cores, CUDA capability 3.5
  - 2x Quadro K4000: 3 GB memory, 768 CUDA cores, CUDA capability 3.0

• 20 single-GPU Machines (listed on the right)
  - Each has one GT 630
  - 2 GB memory, 192 CUDA cores, CUDA capability 3.0

Check [http://report.cs.rutgers.edu/mrtg/systems/ilab.html](http://report.cs.rutgers.edu/mrtg/systems/ilab.html) for the load and availability of different GPU machines
Suggested Reading

• CUDA Programming Guide — Chapter 1 to Chapter 5

https://docs.nvidia.com/cuda/cuda-c-programming-guide/