Lecture 1:

Course Introduction + Review of Throughput Hardware Concepts

Visual Computing Systems
Stanford CS348V : Winter 2018
Hi!

Me: Prof. Kayvon

Your TA: Raj Setaluri
Visual computing applications

2D/3D graphics

Computational photography and image processing

Understanding the contents of images and videos
Visual Computing Systems
— Some History

(why I get so excited about this topic)
Ivan Sutherland’s Sketchpad on MIT TX-2 (1962)
The frame buffer
Shoup’s SuperPaint (PARC 1972-73)

16 2K shift registers (640 x 486 x 8 bits)
The frame buffer
Shoup’s SuperPaint (PARC 1972-73)

16 2K shift registers (640 x 486 x 8 bits)
Xerox Alto (1973)

Bravo (WYSIWYG)

TI 74181 ALU
Goal: render everything you’ve ever seen

“Road to Pt. Reyes”
LucasFilm (1983)
"We take an average of three hours to draw a single frame on the fastest computer money can buy."
- Steve Jobs
UNC Pixel Planes (1981), computation-enhanced frame buffer

ASIC for geometric transforms used in real-time graphics.
Real-time (30 fps) on a NVIDIA Titan X
NVIDIA Titan X Pascal GPU (2017)
(~ 12 TFLOPs fp32)

~ ASCI Q (top US supercomputer circa 2002)
Modern GPU: heterogeneous multi-core

- Multi-threaded, SIMD cores
- Custom circuits for key graphics arithmetic
- Custom circuits for HW-assisted graphics-specific DRAM compression
- HW logic for scheduling work onto these resources
Domain-specific languages for heterogeneous computing

OpenGL Graphics Pipeline (circa 2007)

- **Input vertex buffer**
- **Vertex Generation** → 3D vertex stream
- **Vertex Processing** → Projected vertex stream
- **Primitive Generation** → Primitive stream
- **Fragment Generation** ("Rasterization") → Fragment stream
- **Fragment Processing** → Fragment stream
- **Pixel Operations** → Output image buffer (pixels)
Domain-specific languages for heterogeneous computing

OpenGL Graphics Pipeline (circa 2007)

Input vertex buffer
- Vertex Generation
  - 3D vertex stream
  - Projected vertex stream
- Vertex Processing
- Primitive Generation
  - Primitive stream
- Fragment Generation ("Rasterization")
  - Fragment stream
- Fragment Processing
- Fragment stream
- Pixel Operations
- Output image buffer (pixels)

uniform sampler2D myTexture;
uniform float3 lightDir;
varying vec3 norm;
varying vec2 uv;

void myFragmentShader()
{
  vec3 kd = texture2D(myTexture, uv);
  kd *= clamp(dot(lightDir, norm), 0.0, 1.0);
  return vec4(kd, 1.0);
}

"fragment shader" (a.k.a kernel function mapped onto input fragment stream)

"per-element" inputs
- read-only global variables
- "per-element" inputs

"per-element" output: RGBA surface color at pixel
Emerging state-of-the-art visual computing systems today...

- Intelligent cameras in smartphones
- Cloud servers ("infinite" computing and storage at your disposal as a service)
- Proliferation of specialized compute accelerators
  - For image processing, machine learning
- Proliferation of high-resolution image sensors...
Capturing pixels to communicate

Ingesting/serving the world’s photos

2B photo uploads and shares per day across Facebook sites (incl. Instagram+WhatsApp) [FB2015]

Ingesting/streaming world’s video

YouTube 2015: 300 hours uploaded per minute [Youtube]

Cisco VNI projection: 80-90% of 2019 internet traffic will be video. (64% in 2014)
Richer content: beyond a single image

- Example: Apple’s “Live Photos”
- Each photo is not only a single frame, but a few seconds of video before and after the shutter is clicked.
Facebook Live

Broadcasting live from our red carpet premiere!
VR output
Example: Google’s JumpVR video
Input stream: 16 4K GoPro cameras
Register + 3D align video stream (on edge device)
Broadcast encoded video stream across the country to millions of viewers
High resolution, multi-camera Facebook Surround 360 VR video

2048 x 2048 PointGrey Camera @ 30 FPS

14 cameras
8K x 8K stereo panorama output
VR: high resolution requirements

iPhone 6: 4.7 in “retina” display:
1.3 MPixel
326 ppi → 57 ppd

Future “retina” VR display:
57 ppd covering 180°
= 10K x 10K display per eye
= 200 MPixel

RAW data rate @ 120Hz ≈ 72 GB/sec
Enhancing communication: understanding images to improve acquired content

AutoEnhance:

Photo “fix up” [Hayes 2007]

My bad vacation photo

Part to fix

Portrait Mode:

Similar photos others have taken

Fixed!
On every vehicle: analyzing images for robot navigation
High-resolution video (moving camera)
NVIDIA Drive PX

Tegra X1 (1 TFlop fp16 at 1GHz)
On every corner: analyzing images for urban efficiency

“Managing urban areas has become one of the most important development challenges of the 21st century. Our success or failure in building sustainable cities will be a major factor in the success of the post-2015 UN development agenda.”

- UN Dept. of Economic and Social Affairs
High resolution (static camera)
Sensing human social interactions

CMU Panoptic Studio
480 video cameras (640 x 480 @ 25fps)
116 GPixel video sensor
(2.9 TPixel /sec)

[Joo 2015]
Capturing social interactions

[Courtesy Yaser Sheikh, Tomas Simon, Hanbyul Joo]
Capturing social interactions

[Courtesy Yaser Sheikh, Tomas Simon, Hanbyul Joo]
On every human: analyzing egocentric images to augment humans
AR requires low-latency localization and scene object recognition
Smart headlight system

[Tamburo 2016]

Beamsplitter

Spatial Light Modulator

Camera

Processor

cs.cmu.edu/smartheadlight

~1000 Hz (1 - 1.5 ms latency)
Seeing clearly through precipitation

Idea: Stream Light Between Snowflakes

Goal: High Light Throughput and Accuracy

Illustration adapted from de Charette (ICCP, 2012)
Future challenge: recording and analyzing the world’s visual information, so computers can understand and reason about it
Capturing everything about the visual world

To understand people
To understand the world around vehicles/drones
To understand cities

Mobile
Continuous (always on)
Exceptionally high resolution

Capture for computers to analyze, not humans to watch
What is this course about?

1. Understanding the characteristics of important visual computing workloads
2. Understanding techniques used to achieve efficient system implementations

VISUAL COMPUTING WORKLOADS
Algorithms for 3D graphics, image processing, compression, etc.

PARALLELISM
Exploiting locality
Minimizing communication

DESIGN OF PROGRAMMING ABSTRACTIONS
choice of programming primitives
level of abstraction

MACHINE ORGANIZATION
High-throughput hardware designs:
Parallel, heterogeneous, specialized

OpenGL
TensorFlow
mxnet
In other words

It is about understanding the fundamental structure of problems in the visual computing domain, and then leveraging that understanding to...

To design more efficient algorithms

To build the most efficient hardware to run these applications

To design the right programming systems to make developing new applications simpler, more productive, and highly performant
Course Logistics
Logistics

- Course web site: http://graphics.stanford.edu/courses/cs348v-18-winter
- All announcements will go out via Piazza https://piazza.com/class/jc1f626cfne6r6
- Kayvon’s office hours: Tuesday after class, or by appt.
Expectations of you

- **20% participation**
  - There will be ~1 assigned paper reading per class
  - Everyone is expected to come to class and participate in discussions based on readings
  - You are encouraged discuss papers and or my lectures on the course discussion board.
  - If you form a weekly course reading/study group, I will buy Pizza for said group.

- **30% mini-assignments (3 short programming assignments)**
  - Assignment 1: analyze parallel program performance on a multi-core CPU
  - Assignment 2: implement and optimize a basic RAW image processing pipeline
  - Assignment 3: optimize performance of a modern DNN module

- **20% 1 take-home “exam”**

- **30% self-selected final project**
  - I suggest you start thinking about projects now (can be teams of up to two)
Major course themes/topics

Part 1: High Efficiency Image and Video Processing

Overview of a Modern Digital Camera Processing Pipeline
Image Processing Algorithms You Should Know
Efficiently Scheduling Image Processing Algorithms on Parallel Hardware
Specialized Hardware for Image Processing
Lossy Image (JPG) and Video (H.264) Video Compression
Video Processing/Synthesis for Virtual Reality Display

Part 2: Accelerating Deep Learning for Computer Vision (from a systems perspective)

Workload Characteristics of DNN Inference for Image Analysis
Scheduling and Algorithms for Parallel DNN Training at Scale
A Case Study of Algorithmic Optimizations for Object Detection
Leveraging Task-Specific DNN Structure for Improving Performance and Accuracy
Hardware Accelerators for DNN Inference
Design Space of Dataflow Programming Abstractions for Deep Learning
Enhancing Efficiency Through Model Specialization (in particular for video)
Efficient Inference at Datacenter Scale
Algorithmic innovation in image classification

Improving *accuracy-per-unit cost* using better DNN designs?

2014 → 2017  ~ 25x improvement in cost at similar accuracy

<table>
<thead>
<tr>
<th>Model</th>
<th>ImageNet Top-1 Accuracy</th>
<th>Num Params</th>
<th>Cost/image (MADDs)</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG-16</td>
<td>71.5%</td>
<td>138M</td>
<td>15B</td>
<td>[2014]</td>
</tr>
<tr>
<td>GoogleNet</td>
<td>70%</td>
<td>6.8M</td>
<td>1.5B</td>
<td>[2015]</td>
</tr>
<tr>
<td>ResNet-18</td>
<td>73% *</td>
<td>11.7M</td>
<td>1.8B</td>
<td>[2016]</td>
</tr>
<tr>
<td>MobileNet-224</td>
<td>70.5%</td>
<td>4.2M</td>
<td>0.6B</td>
<td>[2017]</td>
</tr>
</tbody>
</table>

* 10-crop results (ResNet 1-crop results are similar to other DNNs in this table)
Major course themes/topics

Part 3: The GPU Accelerated 3D Graphics Pipeline

Real-Time 3D Graphics Pipeline Architecture
Hardware Acceleration of Z-Buffering and Texturing
Scheduling the Graphics Pipeline onto a GPU
Domain Specific Languages for Shading
Review:
key principles of modern throughput computing hardware
Review concepts

- What are these design concepts, and what problem/goals do they address?
  - Muti-core processing
  - SIMD processing
  - Hardware multi-threading

- What is the motivation for specialization via:
  - Multiple types of processors (e.g., CPUs, GPUs)
  - Custom hardware units (ASIC)

- What is memory bandwidth a major constraint when mapping applications to modern computer systems?
Let’s crack open a modern smartphone

Samsung Galaxy S7 phone with Qualcomm Snapdragon 820 processor

Multi-core GPU
(3D graphics, OpenCL data-parallel compute)

Display engine
(compresses pixels for transfer to 4K screen)

Image Signal Processor (ISP): ASIC for processing pixels off camera (25MP at 30Hz)

Multi-core ARM CPU

Video encode/decode ASIC (H.265 @ 4K)

“Hexagon” Programmable DSP data-parallel multi-media processing
Multi-core processing
Review: what does a processor do?

It runs programs!

Processor executes instruction referenced by the program counter (PC)
(execute the instruction will modify machine state: contents of registers, memory, CPU state, etc.)

Move to next instruction . . .

Then execute it . . .

And so on . . .
Executing an instruction stream

- Fetch/Decode
- ALU (Execute)
- Execution Context

\[
x[i] \\
\]

\[
ld \quad r0, \text{addr}[r1] \\
mul \quad r1, r0, r0 \\
mul \quad r1, r1, r0 \\
... \\
... \\
... \\
... \\
... \\
... \\
... \\
\]

\[
st \quad \text{addr}[r2], r0 \\
\]

result[i]
Executing an instruction stream

My very simple processor: executes one instruction per clock

Fetch/Decode

ALU (Execute)

Execution Context

PC

\[ x[i] \]

\[ ld \ r0, \ addr[r1] \]
\[ mul \ r1, \ r0, \ r0 \]
\[ mul \ r1, \ r1, \ r0 \]
\[ ... \]
\[ ... \]
\[ ... \]
\[ ... \]
\[ ... \]
\[ ... \]
\[ st \ addr[r2], \ r0 \]

result[i]
Executing an instruction stream

My very simple processor: executes one instruction per clock

Fetch/Decode

ALU (Execute)

Execution Context

```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...  
...  
...  
...  
...  
st   addr[r2], r0
```

result[i]
Executing an instruction stream

My very simple processor: executes one instruction per clock

Fetch/Decode

ALU (Execute)

Execution Context

ld r0, addr[r1]
mul r1, r0, r0
mul r1, r1, r0
...
...
...
...
...
...
...
st addr[r2], r0

x[i]

PC

result[i]
Quick aside:
Instruction-level parallelism and superscalar execution
Instruction level parallelism (ILP) example

\[ a = x^2 + y^2 + z^2 \]

Consider the following program:

// assume r0=x, r1=y, r2=z

mul r0, r0, r0
mul r1, r1, r1
mul r2, r2, r2
add r0, r0, r1
add r3, r0, r2

// now r3 stores value of program variable ‘a’

This program has five instructions, so it will take five clocks to execute, correct? Can we do better?
ILP example

\[ a = x^2 + y^2 + z^2 \]
Superscalar execution

\[ a = x^2 + y^2 + z^2 \]

// assume r0=x, r1=y, r2=z

1. mul r0, r0, r0
2. mul r1, r1, r1
3. mul r2, r2, r2
4. add r0, r0, r1
5. add r3, r0, r2

// r3 stores value of variable ‘a’

**Superscalar execution**: processor automatically finds independent instructions in an instruction sequence and executes them in parallel on multiple execution units!

In this example: instructions 1, 2, and 3 can be executed in parallel (on a superscalar processor that determines that the lack of dependencies exists)

But instruction 4 must come after instructions 1 and 2

And instruction 5 must come after instruction 4
Superscalar execution

Program: computes sin of input \( x \) via Taylor expansion

```c
void sinx(int N, int terms, float x) {
    float value = x;
    float numer = x * x * x;
    int denom = 6;  // 3!
    int sign = -1;

    for (int j=1; j<=terms; j++) {
        value += sign * numer / denom;
        numer *= x * x;
        denom *= (2*j+2) * (2*j+3);
        sign *= -1;
    }
    return value;
}
```

My single core, superscalar processor: executes up to two instructions per clock from a single instruction stream.

Independent operations in instruction stream
(They are detected by the processor at run-time and may be executed in parallel on execution units 1 and 2)
Now consider a program that computes the sine of **many** numbers...
Example program

Compute $\sin(x)$ using Taylor expansion: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + ...$

for each element of an array of $N$ floating-point numbers

```c
void sinx(int N, int terms, float* x, float* result)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;  // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```
Multi-core: process multiple instruction streams in parallel

Sixteen cores, sixteen simultaneous instruction streams
Multi-core examples

Intel “Skylake” Core i7 quad-core CPU (2015)

NVIDIA GP104 (GTX 1080) GPU
20 replicated (“SM”) cores (2016)
More multi-core examples


Apple A9 dual-core CPU (2015)

A9 image credit: Chipworks (obtained via Anandtech)
http://www.anandtech.com/show/9686/the-apple-iphone-6s-and-iphone-6s-plus-review/3
SIMD processing
Add ALUs to increase compute capability

Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing
Single instruction, multiple data

Same instruction broadcast to all ALUs
Executed in parallel on all ALUs
void sinx(int N, int terms, float* x, float* result)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;  // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}

Original compiled program:
Processes one array element using scalar instructions on scalar registers (e.g., 32-bit floats)

```
ld  r0, addr[r1]
mul r1, r0, r0
mul r1, r1, r0
...
...
...
...
...
st  addr[r2], r0
```
#include <immintrin.h>

void sinx(int N, int terms, float* x, float* sinx)
{
    float three_fact = 6; // 3!
    for (int i=0; i<N; i+=8)
    {
        __m256 origx = _mm256_load_ps(&x[i]);
        __m256 value = origx;
        __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
        __m256 denom = _mm256_broadcast_ss(&three_fact);
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            // value += sign * numer / denom
            __m256 tmp =
                _mm256_div_ps(_mm256_mul_ps(_mm256_broadcast_ss(sign),numer),denom);
            value = _mm256_add_ps(value, tmp);
            numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
            denom = _mm256_mul_ps(denom, _mm256_broadcast_ss((2*j+2) * (2*j+3)));
            sign *= -1;
        }
        _mm256_store_ps(&sinx[i], value);
    }
}
16 SIMD cores: 128 elements in parallel

16 cores, 128 ALUs, 16 simultaneous instruction streams
Data-parallel expression
(in Kayvon’s fictitious data-parallel language)

```c
void sinx(int N, int terms, float* x, float* result)
{
    // declare independent loop iterations
    forall (int i from 0 to N-1)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;  // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

Compiler understands loop iterations are independent, and that same loop body will be executed on a large number of data elements.

Abstraction facilitates automatic generation of both multi-core parallel code, and vector instructions to make use of SIMD processing capabilities within a core.
What about conditional execution?

(assume logic below is to be executed for each element in input array ‘A’, producing output into the array ‘result’)

```plaintext
float x = A[i];
if (x > 0) {
    float tmp = exp(x,5.f);
    tmp *= kMyConst1;
    x = tmp + kMyConst2;
} else {
    float tmp = kMyConst1;
    x = 2.f * tmp;
}
result[i] = x;
```
What about conditional execution?

(assume logic below is to be executed for each element in input array ‘A’, producing output into the array ‘result’)

```c
float x = A[i];
if (x > 0) {
    float tmp = exp(x,5.f);
    tmp *= kMyConst1;
    x = tmp + kMyConst2;
} else {
    float tmp = kMyConst1;
    x = 2.f * tmp;
}

result[i] = x;
```
Mask (discard) output of ALU

Not all ALUs do useful work!
Worst case: 1/8 peak performance

(assume logic below is to be executed for each element in input array ‘A’, producing output into the array ‘result’)

```c
float x = A[i];
if (x > 0) {
    float tmp = exp(x, 5.f);
    tmp *= kMyConst1;
    x = tmp + kMyConst2;
} else {
    float tmp = kMyConst1;
    x = 2.f * tmp;
}
result[i] = x;
```
After branch: continue at full performance

(alternate logic below is to be executed for each element in input array ‘A’, producing output into the array ‘result’)

```c
if (x > 0) {
    float tmp = exp(x,5.f);
    tmp *= kMyConst1;
    x = tmp + kMyConst2;
} else {
    float tmp = kMyConst1;
    x = 2.f * tmp;
}
```

resume unconditional code

result[i] = x;
Example: eight-core Intel Xeon E5-1660 v4

8 cores
8 SIMD ALUs per core (AVX2 instructions)
490 GFLOPs (@3.2 GHz) (140 Watts)

* Showing only AVX math units, and fetch/decode unit for AVX (additional capability for integer math)
Example: NVIDIA GTX 1080 GPU

20 cores ("SMs")
128 SIMD ALUs per core (@1.6 GHz) = 8.1 TFLOPs (180 Watts)
Part 2: accessing memory
Hardware multi-threading
Terminology

- **Memory latency**
  - The amount of time for a memory request (e.g., load, store) from a processor to be serviced by the memory system
  - Example: 100 cycles, 100 nsec

- **Memory bandwidth**
  - The rate at which the memory system can provide data to a processor
  - Example: 20 GB/s
Stalls

- A processor “stalls” when it cannot run the next instruction in an instruction stream because of a dependency on a previous instruction.

- Accessing memory is a major source of stalls
  
  \[
  \text{ld } r0 \text{ mem}[r2] \\
  \text{ld } r1 \text{ mem}[r3] \\
  \text{add } r0, r0, r1
  \]

  Dependency: cannot execute ‘add’ instruction until data at mem[r2] and mem[r3] have been loaded from memory

- Memory access times ~ 100’s of cycles
  - Memory “access time” is a measure of latency
Review: why do modern processors have caches?

- L1 cache (32 KB)
- L2 cache (256 KB)
- L3 cache (8 MB)

Memory DDR3 DRAM (Gigabytes)

25 GB/sec
Caches reduce length of stalls (reduce latency)

Processors run efficiently when data is resident in caches
Caches reduce memory access latency *

L1 cache (32 KB)
L2 cache (256 KB)
L3 cache (8 MB)

Core 1

Core N

Memory
DDR3 DRAM
(Gigabytes)

25 GB/sec

* Caches also provide high bandwidth data transfer to CPU
Prefetching reduces stalls (hides latency)

- All modern CPUs have logic for prefetching data into caches
  - Dynamically analyze program’s access patterns, predict what it will access soon

- Reduces stalls since data is resident in cache when accessed

```
predict value of r2, initiate load
predict value of r3, initiate load
...
... data arrives in cache
... data arrives in cache
... ld r0 mem[r2]
... ld r1 mem[r3]
add r0, r0, r1
```

Note: Prefetching can also reduce performance if the guess is wrong (hogs bandwidth, pollutes caches)

(more detail later in course)
Multi-threading reduces stalls

- Idea: interleave processing of multiple threads on the same core to hide stalls

- Like prefetching, multi-threading is a latency hiding, not a latency reducing technique
Hiding stalls with multi-threading

Thread 1
Elements 0 … 7

1 Core (1 thread)

Fetch/Decode

ALU 0  ALU 1  ALU 2  ALU 3
ALU 4  ALU 5  ALU 6  ALU 7

Exec Ctx
Hiding stalls with multi-threading

Thread 1
Elements 0 … 7

Thread 2
Elements 8 … 15

Thread 3
Elements 16 … 23

Thread 4
Elements 24 … 31

1 Core (4 hardware threads)
Hiding stalls with multi-threading

Time

Runnable

Stall

Thread 1
Elements 0 … 7

Thread 2
Elements 8 … 15

Thread 3
Elements 16 … 23

Thread 4
Elements 24 … 31

1 Core (4 hardware threads)

1

2

3

4

Fetch/Decode

ALU 0

ALU 1

ALU 2

ALU 3

ALU 4

ALU 5

ALU 6

ALU 7
Hiding stalls with multi-threading

Thread 1
Elements 0 … 7
Runnable

Thread 2
Elements 8 … 15
Runnable

Thread 3
Elements 16 … 23
Runnable

Thread 4
Elements 24 … 31
Runnable

1 Core (4 hardware threads)

Fetch/Decode

ALU 0
ALU 1
ALU 2
ALU 3
ALU 4
ALU 5
ALU 6
ALU 7

1
2
3
4

Stall

Runnable

Done!

Thread 1
Elements 0 … 7
Runnable

Thread 2
Elements 8 … 15
Runnable

Thread 3
Elements 16 … 23
Runnable

Thread 4
Elements 24 … 31
Runnable

Done!
Key idea of throughput-oriented systems:
Potentially increase time to complete work by any one any one thread, in order to increase overall system throughput when running multiple threads.

During this time, this thread is runnable, but it is not being executed by the processor. (The core is running some other thread.)
Kayvon's fictitious multi-core chip

16 cores

8 SIMD ALUs per core
(128 total)

4 threads per core

16 simultaneous instruction streams

64 total concurrent instruction streams

512 independent pieces of work are needed to run chip with maximal latency hiding ability
GPUs: extreme throughput-oriented processors

NVIDIA GTX 1080 core ("SM")

- Instructions operate on 32 pieces of data at a time (instruction streams called "warps").
- Think: warp = thread issuing 32-wide vector instructions
- Different instructions from up to four warps can be executed simultaneously (simultaneous multi-threading)
- Up to 64 warps are interleaved on the SM (interleaved multi-threading)
- Over 2,048 elements can be processed concurrently by a core

Source: NVIDIA Pascal Tuning Guide
There are 20 SM cores on the GTX 1080:
That’s 40,960 pieces of data being processed concurrently to get maximal latency hiding!
Another example: for review and to check your understanding

(if you understand the following sequence you understand this lecture)
Running code on a simple processor

My very simple program: compute $\sin(x)$ using Taylor expansion

```c
void sinx(int N, int terms, float* x, float* result) {
    for (int i=0; i<N; i++) {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;  // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++) {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

My very simple processor: completes one instruction per clock

Fetch/Decode

ALU (Execute)

Execution Context
Review: superscalar execution

Unmodified program

```c
void sinx(int N, int terms, float* x, float* result)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

My single core, superscalar processor: executes up to two instructions per clock from a single instruction stream.

Independent operations in instruction stream
(They are detected by the processor at run-time and may be executed in parallel on execution units 1 and 2)
Review: multi-core execution (two cores)

Modify program to create two threads of control (two instruction streams)

typedef struct {
    int N;
    int terms;
    float* x;
    float* result;
} my_args;

void parallel_sinx(int N, int terms, float* x, float* result) {
    pthread_t thread_id;
    my_args args;

    args.N = N/2;
    args.terms = terms;
    args.x = x;
    args.result = result;

    // launch thread
    pthread_create(&thread_id, NULL, my_thread_start, &args);
    sinx(N - args.N, terms, x + args.N, result + args.N); // do work
    pthread_join(thread_id, NULL);
}

void my_thread_start(void* thread_arg) {
    my_args* thread_args = (my_args*)thread_arg;
    sinx(thread_args->N, thread_args->terms, thread_args->x, thread_args->result); // do work
}

My dual-core processor:
executes one instruction per clock
from an instruction stream on each core.
Review: multi-core + superscalar execution

Modify program to create two threads of control (two instruction streams)

typedef struct {
    int N;
    int terms;
    float* x;
    float* result;
} my_args;

void parallel_sinx(int N, int terms, float* x, float* result) {
    pthread_t thread_id;
    my_args args;

    args.N = N/2;
    args.terms = terms;
    args.x = x;
    args.result = result;

    // launch thread
    pthread_create(&thread_id, NULL, my_thread_start, &args);
    sinx(N - args.N, terms, x + args.N, result + args.N); // do work
    pthread_join(thread_id, NULL);
}

void my_thread_start(void* thread_arg) {
    my_args* thread_args = (my_args*)thread_arg;
    sinx(thread_args->N, thread_args->terms, thread_args->x, thread_args->result); // do work
}

My superscalar dual-core processor: executes up to two instructions per clock from an instruction stream on each core.
Review: multi-core (four cores)

Modify program to create many threads of control:
(code written in Kayvon’s fictitious data-parallel language)

```c
void sinx(int N, int terms, float* x, float* result) {
    // declare independent loop iterations
    forall (int i from 0 to N-1) {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;  // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++) {
            value += sign * numer / denom
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[i] = value;
    }
}
```

My quad-core processor: executes one instruction per clock from an instruction stream on each core.
Review: four, 8-wide SIMD cores

Observation: program must execute many iterations of the same loop body.
Optimization: share instruction stream across execution of multiple iterations (single instruction multiple data = SIMD)

```c
void sinx(int N, int terms, float* x, float* result) {
    // declare independent loop iterations
    forall (int i from 0 to N-1) {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;  // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++) {
            value += sign * numer / denom
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

My SIMD quad-core processor: executes one 8-wide SIMD instruction per clock from an instruction stream on each core.
Review: four SIMD, multi-threaded cores

Observation: memory operations have very long latency
Solution: hide latency of loading data for one iteration by executing arithmetic instructions from other iterations

void sinx(int N, int terms, float* x, float* result)
{
  // declare independent loop iterations
  forall (int i from 0 to N-1)
  {
    float value = x[i];
    float numer = x[i] * x[i] * x[i];
    int denom = 6;  // 3!
    int sign = -1;
    
    for (int j=1; j<=terms; j++)
    {
      value += sign * numer / denom
      numer *= x[i] * x[i];
      denom *= (2*j+2) * (2*j+3);
      sign *= -1;
    }
    result[i] = value;
  }
}
Summary: four superscalar, SIMD, multi-threaded cores

My multi-threaded, superscalar, SIMD quad-core processor:
executes up to two instructions per clock from one instruction stream on each core (in this example: one SIMD instruction + one scalar instruction).
Processor can switch to execute the other instruction stream when faced with stall.
Connecting it all together

Kayvon’s simple quad-core processor:

Four cores, two-way multi-threading per core (max eight threads active on chip at once), up to two instructions per clock per core (one of those instructions is 8-wide SIMD)
Thought experiment

- You write a C application that spawns **two** pthreads
- The application runs on the processor shown below
  - Two cores, two-execution contexts per core, up to instructions per clock, one instruction is an 8-wide SIMD instruction.

- Question: “who” is responsible for mapping your pthreads to the processor’s thread execution contexts?
  
  *Answer: the operating system*

- Question: If you were the OS, how would to assign the two threads to the four available execution contexts?

- Another question: How would you assign threads to execution contexts if your C program spawned **five** pthreads?
Another thought experiment

Task: element-wise multiplication of two vectors A and B

Assume vectors contain millions of elements

- Load input A[i]
- Load input B[i]
- Compute A[i] × B[i]
- Store result into C[i]

Three memory operations (12 bytes) for every MUL

NVIDIA GTX 1080 GPU can do 2560 MULs per clock (@ 1.6 GHz)

Need ~50 TB/sec of bandwidth to keep functional units busy (only have 320 GB/sec)

<1% GPU efficiency... but 4.2x faster than eight-core CPU!

(3.2 GHz Xeon E5v4 eight-core CPU connected to 76 GB/sec memory bus will exhibit ~3% efficiency on this computation)
Bandwidth limited!

If processors request data at too high a rate, the memory system cannot keep up.

No amount of latency hiding helps this.

Bandwidth is a critical resource

Overcoming bandwidth limits are a common challenge for application developers on throughput-optimized systems.
Hardware specialization
Why does energy efficiency matter?

- General mobile processing rule: the longer a task runs the less power it can use.
  - Processor’s power consumption is limited by heat generated (efficiency is required for more than just maximizing battery life).

![Graph showing power consumption over time with labels for Electrical limit, Die temp, Case temp, and Battery life.]

- Electrical limit: max power that can be supplied to chip.
- Die temp: (junction temp -- Tj): chip becomes unreliable above this temp (chip can run at high power for short period of time until chip heats to Tj).
- Case temp: mobile device gets too hot for user to comfortably hold (chip is at suitable operating temp, but heat is dissipating into case).
- Battery life: chip and case are cool, but want to reduce power consumption to sustain long battery life for given task.

---

Slide credit: adopted from original slide from M. Shebanow: HPG 2013 keynote

iPhone 6 battery: 7 watt-hours
9.7in iPad Pro battery: 28 watt-hours
15in Macbook Pro: 99 watt-hours
Efficiency benefits of compute specialization

- Rules of thumb: compared to high-quality C code on CPU...

- Throughput-maximized processor architectures: e.g., GPU cores
  - Approximately 10x improvement in perf / watt
  - Assuming code maps well to wide data-parallel execution and is compute bound

- Fixed-function ASIC ("application-specific integrated circuit")
  - Can approach 100-1000x or greater improvement in perf/watt
  - Assuming code is compute bound and is not floating-point math

[Source: Chung et al. 2010, Dally 08]

[Figure credit Eric Chung]
Hardware specialization increases efficiency

Area-normalized FFT Performance (40nm)

Pseudo-GFLOPs per mm²

100
10
1
0.1

4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

\(\lg_2(N)\) (data set size)

Core i7
LX760
GTX285
GTX480
ASIC

FPGA
GPUs

ASIC delivers same performance as one CPU core with ~ 1/1000th the chip area.

GPU cores: ~ 5-7 times more area efficient than CPU cores.

FFT Energy Efficiency (40nm)

Pseudo-GFLOPs per J

100
10
1
0

4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

\(\lg_2(N)\) (data set size)

Core i7
LX760
GTX285
GTX480
ASIC

FPGA
GPUs

ASIC delivers same performance as one CPU core with only ~ 1/100th the power.

[Chung et al. MICRO 2010]
Modern systems use ASICs for...

- Image/video encode/decode (e.g., H.264, JPG)
- Audio recording/playback
- Voice “wake up” (e.g., Ok Google)
- Camera “RAW” processing: processing data acquired by image sensor into images that are pleasing to humans
- Many 3D graphics tasks (rasterization, texture mapping, occlusion using the Z-buffer)
- Significant modern interest in ASICs for deep network evaluation (e.g., Google’s Tensor Processing Unit)
Qualcomm Hexagon DSP

- Originally used for audio/LTE support on Qualcomm SoC’s
- Multi-threaded, VLIW DSP
- Third major programmable unit on modern Qualcomm SoCs
  - Multi-core CPU
  - Multi-core GPU (Adreno)
  - Hexagon DSP
Summary: choosing the right tool for the job

- Energy-optimized CPU
- Throughput-oriented processor (GPU)
- Programmable DSP
- FPGA/Future reconfigurable logic
- ASIC

- Video encode/decode, Audio playback, Camera RAW processing, neural nets (future?)

~10X more efficient
~100X???
~100-1000X

Easiest to program
Difficult to program (making it easier is active area of research)
Not programmable + costs 10-100’s millions of dollars to design / verify / create

Credit Pat Hanrahan for this taxonomy
Data movement has high energy cost

- **Rule of thumb in mobile system design: always seek to reduce amount of data transferred from memory**
  - Earlier in class we discussed minimizing communication to reduce stalls (poor performance). Now, we wish to reduce communication to reduce energy consumption

- **“Ballpark” numbers** [Sources: Bill Dally (NVIDIA), Tom Olson (ARM)]
  - Integer op: ~1 pJ*
  - Floating point op: ~20 pJ*
  - Reading 64 bits from small local SRAM (1mm away on chip): ~26 pJ
  - Reading 64 bits from low power mobile DRAM (LPDDR): ~1200 pJ

- **Implications**
  - Reading 10 GB/sec from memory: ~1.6 watts
  - Entire power budget for mobile GPU: ~1 watt
    (remember phone is also running CPU, display, radios, etc.)
  - iPhone 6 battery: ~7 watt-hours (note: my Macbook Pro laptop: 99 watt-hour battery)
  - Exploiting locality matters!!!

*Cost to just perform the logical operation, not counting overhead of instruction decode, load data from registers, etc.*
Welcome to cs348v!

- Make sure you are signed up on Piazza so you get announcements

- Tonight’s reading:
More review
For the rest of this class, know these terms

- Multi-core processor
- SIMD execution
- Coherent control flow
- Hardware multi-threading
  - Interleaved multi-threading
  - Simultaneous multi-threading
- Memory latency
- Memory bandwidth
- Bandwidth bound application
- Arithmetic intensity
Which program performs better?

Program 1

```c
void add(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
        C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
        C[i] = A[i] * B[i];
}


// assume arrays are allocated here

// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

Program 2

```c
void fused(int n, float* A, float* B, float* C, float* D, float* E) {
    for (int i=0; i<n; i++)
        E[i] = D[i] + (A[i] + B[i]) * C[i];
}

// compute E = D + (A + B) * C
fused(n, A, B, C, D, E);
```
More thought questions

Program 1

```c
void add(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
        C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
        C[i] = A[i] * B[i];
}

// assume arrays are allocated here

// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

Program 2

```c
void fused(int n, float* A, float* B, float* C, float* D, float* E) {
    for (int i=0; i<n; i++)
        E[i] = D[i] + (A[i] + B[i]) * C[i];
}

// compute E = D + (A + B) * C
fused(n, A, B, C, D, E);
```

Which code structuring style would you rather write?

Consider running either of these programs: would CPU support for hardware-multi-threading help performance?
Visualizing interleaved and simultaneous multi-threading (and combinations thereof)
Interleaved multi-threading

Consider a processor with:
- Two execution contexts
- One fetch and decode unit (one instruction per clock)
- One ALU (to execute the instruction)

In an interleaved multi-threading scenario, the threads share the processor.
(This is a visualization of when threads are having their instructions executed by the ALU.)
Interleaved multi-threading

Consider a processor with:
- Two execution contexts
- One fetch and decode unit (one instruction per clock)
- One ALU (to execute the instruction)

Same as previous slide, but now just a different scheduling order of the threads (fine-grained interleaving)
Simultaneous multi-threading

Consider a processor with:
- Two execution contexts
- Two fetch and decode units (two instructions per clock)
- Two ALUs (to execute the two instructions)

In an simultaneous multi-threading scenario, the threads execute simultaneously on the two ALUs. (note, no ILP in a thread since each thread is run sequentially on one ALU)
Combining simultaneous and interleaved multi-threading

Consider a processor with:

- **Four execution contexts**
- Two fetch and decode units (two instructions per clock, choose two of four threads)
- Two ALUs (to execute the two instructions)

![Diagram showing execution contexts and ALUs](image)
Another way to visualize execution (ALU-centric view)

Consider a processor with:
- Four execution contexts
- Two fetch and decode units (two instructions per clock, choose two of four threads)
- Two ALUs (to execute the two instructions)

Now the graph is visualizing what each ALU is doing each clock:
Instructions can be drawn from same thread (ILP)

Consider a processor with:
- Four execution contexts
- Two fetch and decode units (two instructions per clock, choose any two independent instructions from the four threads)
- Two ALUs (to execute the two instructions)

![Diagram showing time (clocks) with ALU 0 and ALU 1, and two instructions from same thread executing simultaneously.]

- Blue = executing T0 at this time
- Light blue = executing T1 at this time
- Green = executing T2 at this time
- Yellow = executing T3 at this time