Lecture 2:
A Modern Multi-Core Processor
(Forms of parallelism + understanding latency and bandwidth)

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Fall 2018
Today

- Today we will talk computer architecture

- Four key concepts about how modern computers work
  - Two concern parallel execution
  - Two concern challenges of accessing memory

- Understanding these architecture basics will help you
  - Understand and optimize the performance of your parallel programs
  - Gain intuition about what workloads might benefit from fast parallel machines
Part 1: parallel execution
Example program

Compute \( \sin(x) \) using Taylor expansion: \( \sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots \) 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;
    }
}
```
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;
    }
}
Execute program

Fetch/Decode

ALU (Execute)

Execution Context

ld r0, addr[r1]
mul r1, r0, r0
mul r1, r1, r0
...
...
...
...
...
...
...
result[i]
Execute program

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]
Execute program

My very simple processor: executes one instruction per clock

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

result[i]

PC
Execute program

My very simple processor: executes one instruction per clock

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

result[i]
Superscalar processor

Recall from last class: instruction level parallelism (ILP)
Decode and execute two instructions per clock (if possible)

Note: No ILP exists in this region of the program
Aside: Pentium 4

Processor: pre multi-core era

Majority of chip transistors used to perform operations that help a single instruction stream run fast

- Fetch/Decode
- ALU (Execute)
- Execution Context
- Data cache (a big one)
- Out-of-order control logic
- Fancy branch predictor
- Memory pre-fetcher

More transistors = larger cache, smarter out-of-order logic, smarter branch predictor, etc.
(Also: more transistors $\rightarrow$ smaller transistors $\rightarrow$ higher clock frequencies)
Processor: multi-core era

Idea #1:

Use increasing transistor count to add more cores to the processor

Rather than use transistors to increase sophistication of processor logic that accelerates a single instruction stream (e.g., out-of-order and speculative operations)
Two cores: compute two elements in parallel

Simpler cores: each core is slower at running a single instruction stream than our original “fancy” core (e.g., 0.75 times as fast)

But there are now two cores: $2 \times 0.75 = 1.5$ (potential for speedup!)
But our program expresses no parallelism

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;
    }
}
Expressing parallelism using pthreads

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

def sin(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;
    }
}

def parallel_sin(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;

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

def 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
}
Data-parallel expression

(in our 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;
    }
}
```

Loop iterations declared by the programmer to be independent

With this information, you could imagine how a compiler might automatically generate parallel threaded code.
Four cores: compute four elements in parallel
Sixteen cores: compute sixteen elements in parallel

Sixteen cores, sixteen simultaneous instruction streams
Multi-core examples

Intel “Coffee Lake” Core i7 hexa-core CPU (2017)

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

Intel Xeon Phi “Knights Landing” 76-core CPU
(2015)

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
**Data-parallel expression**

*(in our 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;
    }
}
```

Another interesting property of this code:

Parallelism is across iterations of the loop.

All the iterations of the loop do the same thing: evaluate the sine of a single input number.
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
Add ALUs to increase compute capability

Recall original compiled program:

Instruction stream processes one array element at a time using scalar instructions on scalar registers (e.g., 32-bit floats)
Scalar program

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
```
Vector program (using AVX intrinsics)

```c
#include <immintrin.h>

void sinx(int N, int terms, float* x, float* result) {
    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_set1ps(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(&result[i], value);
    }
}
```

Intrinsics available to C programmers
Vector program (using AVX intrinsics)

```c
#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_mul_ps(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);
    }
}
```

Compiled program:
Processes eight array elements simultaneously using vector instructions on 256-bit vector registers
16 SIMD cores: 128 elements in parallel

16 cores, 128 ALUs, 16 simultaneous instruction streams
Data-parallel expression
(in our 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?

```
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?

Time (clocks)

1 2 ... 8
ALU 1 ALU 2 ... ALU 8

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

```cpp
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;
}
<resume unconditional code>
result[i] = x;
```
Mask (discard) output of ALU

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

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

```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;
```
Terminology

- **Instruction stream coherence** ("coherent execution")
  - Same instruction sequence applies to all elements operated upon simultaneously
  - Coherent execution is necessary for efficient use of SIMD processing resources
  - Coherent execution IS NOT necessary for efficient parallelization across cores, since each core has the capability to fetch/decode a different instruction stream

- "**Divergent**" execution
  - A lack of instruction stream coherence

- **Note:** don’t confuse instruction stream coherence with "cache coherence" (a major topic later in the course)
SIMD execution on modern CPUs

- **SSE instructions**: 128-bit operations: 4x32 bits or 2x64 bits (4-wide float vectors)
- **AVX instructions**: 256 bit operations: 8x32 bits or 4x64 bits (8-wide float vectors)

- Instructions are generated by the compiler
  - Parallelism explicitly requested by programmer using intrinsics
  - Parallelism conveyed using parallel language semantics (e.g., `forall` example)
  - Parallelism inferred by dependency analysis of loops (hard problem, even best compilers are not great on arbitrary C/C++ code)

- Terminology: “explicit SIMD”: SIMD parallelization is performed at compile time
  - Can inspect program binary and see instructions (`vstoreps, vmulps, etc.`)
SIMD execution on many modern GPUs

- "Implicit SIMD"
  - Compiler generates a scalar binary (scalar instructions)
  - But N instances of the program are *always run* together on the processor
    `execute(my_function, N) // execute my_function N times`
  - In other words, the interface to the hardware itself is data-parallel
  - Hardware (not compiler) is responsible for simultaneously executing the same
    instruction from multiple instances on different data on SIMD ALUs

- SIMD width of most modern GPUs ranges from 8 to 32
  - Divergence can be a big issue
    (poorly written code might execute at 1/32 the peak capability of the machine!)
Example: Intel Core i7

- 4 cores
- 8 SIMD ALUs per core
- (AVX instructions)

On campus:

GHC machines:
- 4 cores
- 8 SIMD ALUs per core

Machines in GHC 5207:
- (old GHC 3000 machines)
- 6 cores
- 4 SIMD ALUs per core

CPUs in "latedays" cluster:
- 6 cores
- 8 SIMD ALUs per code
Example: NVIDIA GTX 480 (in the Gates 5 lab)

15 cores
32 SIMD ALUs per core
1.3 TFLOPS
Summary: parallel execution

Several forms of parallel execution in modern processors

- Multi-core: use multiple processing cores
  - Provides thread-level parallelism: simultaneously execute a completely different instruction stream on each core
  - Software decides when to create threads (e.g., via pthreads API)

- SIMD: use multiple ALUs controlled by same instruction stream (within a core)
  - Efficient design for data-parallel workloads: control amortized over many ALUs
  - Vectorization can be done by compiler (explicit SIMD) or at runtime by hardware
  - [Lack of] dependencies is known prior to execution (usually declared by programmer, but can be inferred by loop analysis by advanced compiler)

- Superscalar: exploit ILP within an instruction stream. Process different instructions from the same instruction stream in parallel (within a core)
  - Parallelism automatically and dynamically discovered by the hardware during execution (not programmer visible)

Not addressed further in this class. That’s for a proper computer architecture design course like 18-447.
Part 2: accessing memory
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

  ```
  ld r0 mem[r2]
  ld r1 mem[r3]
  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?
Caches reduce length of stalls (reduce latency)

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

* 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
...
...
...
...
...
...

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)

(These loads are cache hits)

(data arrives in cache)

(data arrives in cache)

(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

Thread 1
Elements 0 … 7

Thread 2
Elements 8 … 15

Thread 3
Elements 16 … 23

Thread 4
Elements 24 … 31

Runnable

Time

Stall

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

1 Core (4 hardware threads)

Fetch/Decode

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

Thread 1
Elements 0 ... 7

Thread 2
Elements 8 ... 15

Thread 3
Elements 16 ... 23

Thread 4
Elements 24 ... 31

Runnable

Stall

Done!

Elements 0 ... 7

Elements 8 ... 15

Elements 16 ... 23

Elements 24 ... 31

Time
Throughput computing trade-off

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.)
Storing execution contexts

Consider on ship storage of execution contexts a finite resource.

Context storage (or L1 cache)
Many small contexts (high latency hiding ability)

1 core
(16 hardware threads, storage for small working set per thread)
Four large contexts (low latency hiding ability)

1 core
(4 hardware threads, storage for larger working set per thread)
Hardware-supported multi-threading

- Core manages execution contexts for multiple threads
  - Runs instructions from runnable threads (processor makes decision about which thread to run each clock, not the operating system)
  - Core still has the same number of ALU resources: multi-threading only helps use them more efficiently in the face of high-latency operations like memory access

- Interleaved multi-threading (a.k.a. temporal multi-threading)
  - What I described on the previous slides: each clock, the core chooses a thread, and runs an instruction from the thread on the ALUs

- Simultaneous multi-threading (SMT)
  - Each clock, core chooses instructions from multiple threads to run on ALUs
  - Extension of superscalar CPU design
  - Example: Intel Hyper-threading (2 threads per core)
Multi-threading summary

- **Benefit:** use a core’s ALU resources more efficiently
  - Hide memory latency
  - Fill multiple functional units of superscalar architecture
    (when one thread has insufficient ILP)

- **Costs**
  - Requires additional storage for thread contexts
  - Increases run time of any single thread
    (often not a problem, we usually care about throughput in parallel apps)
  - Requires additional independent work in a program (more independent work than ALUs!)
  - Relies heavily on memory bandwidth
    - More threads → larger working set → less cache space per thread
    - May go to memory more often, but can hide the latency
Our 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 480 core

- Fetch/Decode
- Execution contexts (128 KB)
- “Shared” memory (16+48 KB)

= SIMD function unit, control shared across 16 units (1 MUL-ADD per clock)

- Instructions operate on 32 pieces of data at a time (called “warps”).
- Think: warp = thread issuing 32-wide vector instructions
- Up to 48 warps are simultaneously interleaved
- Over 1500 elements can be processed concurrently by a core

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G
NVIDIA GTX 480: more detail (just for the curious)

NVIDIA GTX 480 core

- Fetch/Decode
- SIMD function unit, control shared across 16 units (1 MUL-ADD per clock)

- Execution contexts (128 KB)

- “Shared” memory (16+48 KB)

- Why is a warp 32 elements and there are only 16 SIMD ALUs?

- It’s a bit complicated: ALUs run at twice the clock rate of rest of chip. So each decoded instruction runs on 32 pieces of data on the 16 ALUs over two ALU clocks. (but to the programmer, it behaves like a 32-wide SIMD operation)

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G
NVIDIA GTX 480: more detail (just for the curious)

NVIDIA GTX 480 core

- = SIMD function unit, control shared across 16 units (1 MUL-ADD per clock)

- This process occurs on another set of 16 ALUs as well

- So there are 32 ALUs per core

- 15 cores $\times$ 32 = 480 ALUs per chip

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G
Recall, there are 15 cores on the GTX 480:
That’s 23,000 pieces of data being processed concurrently!
**CPU vs. GPU memory hierarchies**

**CPU:**
- Big caches, few threads, modest memory BW
- Rely mainly on caches and prefetching

**GPU:**
- Small caches, many threads, huge memory BW
- Rely mainly on multi-threading

---

**Memory**
- DDR3 DRAM
  - (Gigabytes)
- DDR5 DRAM
  - (~1 GB)

---

**Core 1**
- L1 cache (32 KB)
- L2 cache (256 KB)
- L3 cache (8 MB)
- Execution contexts (128 KB)

**Core N**
- L1 cache (32 KB)
- L2 cache (256 KB)
- Execution contexts (128 KB)

---

**Execution contexts**
- (128 KB)

---

**Scratchpad L1 cache**
- (64 KB)

---

25 GB/sec

---

177 GB/sec
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 480 GPU can do 480 MULs per clock (@ 1.2 GHz)

Need ~6.4 TB/sec of bandwidth to keep functional units busy (only have 177 GB/sec)

~ 3% efficiency... but 7x faster than quad-core CPU!

(2.6 GHz Core i7 Gen 4 quad-core CPU connected to 25 GB/sec memory bus will exhibit similar 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.

Overcoming bandwidth limits are a common challenge for application developers on throughput-optimized systems.
Bandwidth is a critical resource

Performant parallel programs will:

- Organize computation to fetch data from memory less often
  - Reuse data previously loaded by the same thread (traditional intra-thread temporal locality optimizations)
  - Share data across threads (inter-thread cooperation)

- Request data less often (instead, do more arithmetic: it’s “free”)
  - Useful term: “arithmetic intensity” — ratio of math operations to data access operations in an instruction stream
  - Main point: programs must have high arithmetic intensity to utilize modern processors efficiently
Summary

- Three major ideas that all modern processors employ to varying degrees:
  - Employ multiple processing cores
    - Simpler cores (embrace thread-level parallelism over instruction-level parallelism)
  - Amortize instruction stream processing over many ALUs (SIMD)
    - Increase compute capability with little extra cost
  - Use multi-threading to make more efficient use of processing resources (hide latencies, fill all available resources)

- Due to high arithmetic capability on modern chips, many parallel applications (on both CPUs and GPUs) are bandwidth bound

- GPU architectures use the same throughput computing ideas as CPUs: but GPUs push these concepts to extreme scales
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
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

![Diagram of processor stages: 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;

    pthread_create(&thread_id, NULL, my_thread_start, &args); // launch thread
    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.
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_sin_x(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;

    pthread_create(&thread_id, NULL, my_thread_start, &args); // launch thread
    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
}
Review: multi-core (four cores)

Modify program to create many threads of control: recall our fictitious 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)

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

```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;
    }
}
```
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

Our 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?