Lecture 3:

A Modern Multi-Core Processor
(Forms of parallelism + understanding latency and bandwidth)
Quick review

1. Why has single-instruction-stream performance only improved very slowly in recent years? *

2. What prevented us from obtaining maximum speedup from the parallel programs we performed in the first lecture?

* Self check 1: What do I mean by “single-instruction stream”?

Self check 2: When we talked about the optimization of superscalar execution, were we talking about optimizing the performance of executing a single-instruction stream?
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: 
\[
\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...
\]
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 (on an idealized machine)

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

result[i]

x[i]
Execute program

My very simple processor: executes one instruction per clock

![Diagram showing the execution process]

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

Example instructions:
- `ld r0, addr[r1]`
- `mul r1, r0, r0`
- `mul r1, r1, r0`
- ...
- ...
- ...
- ...
- ...
- ...
- `st addr[r2], r0`

Result:
- `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]
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 (in-order) 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

More transistors = larger cache, smarter out-of-order logic, smarter branch predictor, etc.

(Also: more transistors → smaller transistors → 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., 25% slower)

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

```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;
    }
}
```

This program, compiled with gcc will run as one thread on one of the processor cores.

If each of the simpler processor cores was 25% slower than the original single complicated one, our program now runs 25% slower. :-(
**Expressing parallelism using pthreads**

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

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;
    }
}

typeid 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
}

Child thread does first half of range
Main thread does second half
```
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 “Skylake” Core i7 quad-core CPU (2015)

Each core is sophisticated, out-of-order processor to maximize ILP

NVIDIA GTX 980 GPU
16 replicated processing cores ("SM") (2014)

Each core processors vectors of data
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
Simultaneous Multi-Threading (Hyperthreading)

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

Single core does the work of multiple cores
- Fetch/decode independent instruction streams from different threads
- Map onto shared out-of-order execution unit
- Make fuller use of execution unit when ILP is low
- Reduce impact of stalls / mispredictions
Another View of SMT

Replicate in-order resources
Share out-of-order resources

Does not improve (or penalize) single-thread performance
May get improved multi-threaded performance
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 main loop do the same thing: compute the sine of a single input number

The inner loop is executed the same number of times every time

There are no conditionals
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

```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;
    }
}
```

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)

#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_set1ps(three_fact);
        float 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_set1ps((2*j+2) * (2*j+3)));
            sign *= -1;
        }
        _mm256_store_ps(&result[i], value);
    }
}
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_set1ps(three_fact);
        float 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_set1ps((2*j+2) * (2*j+3)));
            sign *= -1;
        }
        _mm256_store_ps(&result[i], value);
    }
}
```

Compiled program:
Processes eight array elements simultaneously using vector instructions on 256-bit vector registers

\[ 256 = 8 \times 32 \]
What about conditional execution?

(alternate 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;
}

<resume unconditional code>
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`)

```
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

Time (clocks)

ALU 1 ALU 2 ... ALU 8

1 2 ... 8

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

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

(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;
```
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)
- **AVX512 instructions**: 512-bit operations: 16x32 bits or 8x64 bits (16-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!)
SIMD + Multi-core

16 cores, 128 ALUs, 16 simultaneous instruction streams
Example: Intel Core i7

- 4 cores
- 8 SIMD ALUs per core
- AVX instructions
- 32-bit floats

On campus:
- GHC machines:
  - 8 cores
  - 8 SIMD ALUs per core
- Latedays cluster:
  - 12 cores
  - 8 SIMD ALUs per core
Example: NVIDIA GTX 1080

- 20 cores
- 32 SIMD ALUs per core
- 8.2 TFLOPS

(in the Gates 5 lab)
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: parallel execution

Several forms of parallel execution in modern processors

- Multi-core: use multiple processing cores
  - Simultaneous multithreading: Have single core handle multiple threads at same time
    - 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)
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?

- Core 1
  - L1 cache (32 KB)
  - L2 cache (256 KB)

- Core N
  - L1 cache (32 KB)
  - L2 cache (256 KB)

- L3 cache (8 MB)

- Memory
  - DDR3 DRAM
  - 25 GB/sec
  - (Gigabytes)
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

```plaintext
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)

(more detail later in course)
Prefetching in Action

- Two levels of cache
- No prefetching
- Big drop-off in performance for stride=1 as fall out of L1 cache

- Three levels of cache
- Prefetching
- Stride=1 performance high as long as fit into L3 cache
Multi-threading reduces stalls

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

- Helps even with in-order processing of instructions

- Like prefetching, multi-threading is a latency **hiding**, not a latency **reducing** technique

- We will consider several forms of multi-threading
Hiding stalls with Multi-Threading

1 Core (1 thread)

Fetch/Decode

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

Exec Ctx

Thread 1
Elements 0 … 7

Time
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)

Fetch/Decode

ALU 0
ALU 1
ALU 2
ALU 3
ALU 4
ALU 5
ALU 6
ALU 7
Hiding stalls with Multi-Threading

1 Core (4 hardware threads)

Thread 1
Elements 0 … 7

Thread 2
Elements 8 … 15

Thread 3
Elements 16 … 23

Thread 4
Elements 24 … 31

Runnable

Fetch/Decode

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

1

2

3

4

Stall
Hiding stalls with Multi-Threading

Thread 1
Elements 0 … 7
Runnable
Done!

Thread 2
Elements 8 … 15
Runnable
Stall
Runnable

Thread 3
Elements 16 … 23
Runnable
Stall
Runnable

Thread 4
Elements 24 … 31
Runnable
Stall
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
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-chip storage of execution contexts a finite resource.
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)
  - As 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)
  - Extension of out-of-order CPU design
  - Example: Intel Hyper-threading (2 threads per core)
Multi-threading summary

- **Benefit:** use a core’s execution 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

- **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

- $48 \times 32 = 1536$ 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)

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)**

<table>
<thead>
<tr>
<th>Fetch/Decode</th>
<th>Fetch/Decode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution contexts (128 KB)</td>
<td>“Shared” memory (16+48 KB)</td>
</tr>
</tbody>
</table>

- **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 × 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 $48 \times 32 \times 15 = 23,040$ 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
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
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_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
}
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)

```
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

```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 multi-threaded, SIMD quad-core processor: executes one SIMD instruction per clock from one instruction stream on each core. But can switch to processing the other instruction stream when faced with a stall.
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?