Lecture 2:

A Modern Multi-Core Processor

Parallel Computing
Stanford CS149, Fall 2023
Review from class 1:
What is a computer program?
A program is a list of processor instructions!

```c
int main(int argc, char** argv) {
    int x = 1;
    for (int i=0; i<10; i++) {
        x = x + x;
    }
    printf("%d\n", x);
    return 0;
}
```

Compile code

```
_main:
100000f10:  pushq  %rbp
100000f11:  movq %rsp, %rbp
100000f14:  subq  $32, %rsp
100000f18:  movl  $0, -4(%rbp)
100000f1f:  movl %edi, -8(%rbp)
100000f22:  movq %rsi, -16(%rbp)
100000f26:  movl  $1, -20(%rbp)
100000f2d:  movl  $0, -24(%rbp)
100000f34:  cmpl  $10, -24(%rbp)
100000f38:  jge  23 <_main+0x45>
100000f3e:  movl  -20(%rbp), %eax
100000f41:  addl  -20(%rbp), %eax
100000f44:  movl %eax, -20(%rbp)
100000f47:  movl  -24(%rbp), %eax
100000f4a:  addl  $1, %eax
100000f4d:  movl %eax, -24(%rbp)
100000f50:  jmp  -33 <_main+0x24>
100000f55:  leaq  58(%rip), %rdi
100000f5c:  movl  -20(%rbp), %esi
100000f61:  callq  14
100000f66:  xorl %esi, %esi
100000f68:  movl %eax, -28(%rbp)
100000f6b:  movl %esi, %eax
100000f6d:  addq  $32, %rsp
100000f71:  popq %rbp
100000f72:  rets
```
Review from class 1: What does a processor do?
A processor executes instructions

Professor Kayvon’s Very Simple Processor

- **Fetch/Decode**: Determine what instruction to run next
- **ALU (Execution Unit)**: Execution unit: performs the operation described by an instruction, which may modify values in the processor’s registers or the computer’s memory
- **Execution Context**: Registers: maintain program state: store value of variables used as inputs and outputs to operations

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register 0 (R0)</td>
<td></td>
</tr>
<tr>
<td>Register 1 (R1)</td>
<td></td>
</tr>
<tr>
<td>Register 2 (R2)</td>
<td></td>
</tr>
<tr>
<td>Register 3 (R3)</td>
<td></td>
</tr>
</tbody>
</table>
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
```
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
```
Execute program

My very simple processor: executes one instruction per clock
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
```
A program with instruction level parallelism

Program (sequence of instructions)

<table>
<thead>
<tr>
<th>PC</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>a = 2</td>
</tr>
<tr>
<td>01</td>
<td>b = 4</td>
</tr>
<tr>
<td>02</td>
<td>tmp2 = a + b</td>
</tr>
<tr>
<td>03</td>
<td>tmp3 = tmp2 + a</td>
</tr>
<tr>
<td>04</td>
<td>tmp4 = b + b</td>
</tr>
<tr>
<td>05</td>
<td>tmp5 = b * b</td>
</tr>
<tr>
<td>06</td>
<td>tmp6 = tmp2 + tmp4</td>
</tr>
<tr>
<td>07</td>
<td>tmp7 = tmp5 + tmp6</td>
</tr>
<tr>
<td>08</td>
<td>if (tmp3 &gt; 7)</td>
</tr>
<tr>
<td>09</td>
<td>print tmp3</td>
</tr>
<tr>
<td>10</td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>print tmp7</td>
</tr>
</tbody>
</table>

Instruction dependency graph

value during execution
Superscalar processor

This processor can decode and execute up to two instructions per clock.

- **Out-of-order control logic**
- **Fetch/Decode 1**
- **Fetch/Decode 2**
- **Exec 1**
- **Exec 2**

**Superscalar execution**: processor automatically finds independent instructions in an instruction sequence and can execute them in parallel on multiple execution units.

What does it mean for a superscalar processor to “respect program order”?
Review from class 1:
What is memory?
A computer’s memory is organized as an array of bytes

Each byte is identified by its “address” in memory (its position in this array)
(We’ll assume memory is byte-addressable)

“The byte stored at address 0x8 has the value 32.”

“The byte stored at address 0x10 (16) has the value 128.”

In the illustration on the right, the program’s memory address space is 32 bytes in size
(so valid addresses range from 0x0 to 0x1F)
What are caches?

- A cache is a hardware implementation detail that does not impact the output of a program, only its performance.
- Cache is on-chip storage that maintains a copy of a subset of the values in memory.
- If an address is stored “in the cache” the processor can load/store to this address more quickly than if the data resides only in DRAM.
- Caches operate at the granularity of “cache lines”.

In the figure, the cache:
- Has a capacity of 2 lines.
- Each line holds 4 bytes of data.
Cache example 1

Array of 16 bytes in memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>16</td>
</tr>
<tr>
<td>0x1</td>
<td>255</td>
</tr>
<tr>
<td>0x2</td>
<td>14</td>
</tr>
<tr>
<td>0x3</td>
<td>0</td>
</tr>
<tr>
<td>0x4</td>
<td>0</td>
</tr>
<tr>
<td>0x5</td>
<td>0</td>
</tr>
<tr>
<td>0x6</td>
<td>6</td>
</tr>
<tr>
<td>0x7</td>
<td>0</td>
</tr>
<tr>
<td>0x8</td>
<td>32</td>
</tr>
<tr>
<td>0x9</td>
<td>48</td>
</tr>
<tr>
<td>0xA</td>
<td>255</td>
</tr>
<tr>
<td>0xB</td>
<td>255</td>
</tr>
<tr>
<td>0xC</td>
<td>255</td>
</tr>
<tr>
<td>0xD</td>
<td>0</td>
</tr>
<tr>
<td>0xE</td>
<td>0</td>
</tr>
<tr>
<td>0xF</td>
<td>0</td>
</tr>
</tbody>
</table>

Assume:
- Total cache capacity of 8 bytes
- Cache with 4-byte cache lines (So 2 lines fit in cache)
- Least recently used (LRU) replacement policy

Cache state (after load is complete)

<table>
<thead>
<tr>
<th>Address accessed</th>
<th>Cache action</th>
<th>Cache state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>“cold miss”, load 0x0</td>
<td>0x0</td>
</tr>
<tr>
<td>0x1</td>
<td>hit</td>
<td>0x0</td>
</tr>
<tr>
<td>0x2</td>
<td>hit</td>
<td>0x0</td>
</tr>
<tr>
<td>0x3</td>
<td>hit</td>
<td>0x0</td>
</tr>
<tr>
<td>0x4</td>
<td>“cold miss”, load 0x4</td>
<td>0x0</td>
</tr>
<tr>
<td>0x5</td>
<td>hit</td>
<td>0x0</td>
</tr>
<tr>
<td>0x6</td>
<td>hit</td>
<td>0x0</td>
</tr>
<tr>
<td>0x7</td>
<td>hit</td>
<td>0x0</td>
</tr>
<tr>
<td>0x8</td>
<td>0x4</td>
<td>0x4</td>
</tr>
<tr>
<td>0x9</td>
<td>hit</td>
<td>0x0</td>
</tr>
<tr>
<td>0xA</td>
<td>hit</td>
<td>0x0</td>
</tr>
<tr>
<td>0xB</td>
<td>hit</td>
<td>0x0</td>
</tr>
<tr>
<td>0xC</td>
<td>0x4</td>
<td>0x4</td>
</tr>
<tr>
<td>0xD</td>
<td>0x0</td>
<td>0x0</td>
</tr>
<tr>
<td>0xE</td>
<td>0x4</td>
<td>0x4</td>
</tr>
<tr>
<td>0xF</td>
<td>0x0</td>
<td>0x0</td>
</tr>
</tbody>
</table>

There are two forms of “data locality” in this sequence:

Spatial locality: loading data in a cache line “preloads” the data needed for subsequent accesses to different addresses in the same line, leading to cache hits

Temporal locality: repeated accesses to the same address result in hits.
Cache example 2

Array of 16 bytes in memory

Assume:
- Total cache capacity of 8 bytes
- Cache with 4-byte cache lines (So 2 lines fit in cache)
- Least recently used (LRU) replacement policy

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>16</td>
</tr>
<tr>
<td>0x1</td>
<td>255</td>
</tr>
<tr>
<td>0x2</td>
<td>14</td>
</tr>
<tr>
<td>0x3</td>
<td>0</td>
</tr>
<tr>
<td>0x4</td>
<td>0</td>
</tr>
<tr>
<td>0x5</td>
<td>0</td>
</tr>
<tr>
<td>0x6</td>
<td>6</td>
</tr>
<tr>
<td>0x7</td>
<td>0</td>
</tr>
<tr>
<td>0x8</td>
<td>32</td>
</tr>
<tr>
<td>0x9</td>
<td>48</td>
</tr>
<tr>
<td>0xA</td>
<td>255</td>
</tr>
<tr>
<td>0xB</td>
<td>255</td>
</tr>
<tr>
<td>0xC</td>
<td>255</td>
</tr>
<tr>
<td>0xD</td>
<td>0</td>
</tr>
<tr>
<td>0xE</td>
<td>0</td>
</tr>
<tr>
<td>0xF</td>
<td>0</td>
</tr>
</tbody>
</table>

Cache action

- **Line 0x0**: 0x0 “cold miss”, load 0x0
- **Line 0x4**: 0x0 hit
- **Line 0x8**: 0x0 hit
- **Line 0xC**: 0x0 hit
- **Line 0x0**: 0x4 “cold miss”, load 0x4
- **Line 0x0**: 0x8 “cold miss”, load 0x8 (evict 0x0)
- **Line 0xC**: 0x8 “cold miss”, load 0xC (evict 0x4)
- **Line 0x0**: 0xC “capacity miss”, load 0x0 (evict 0x8)

Cache state (after load is complete)

- **0x0**: 0x0 ●●●●
- **0x4**: 0x4 ●●●●
- **0x8**: 0x8 ●●●●
- **0xC**: 0xC ●●●●
Caches reduce length of stalls
(reduce memory access latency)

- Processors run efficiently when they access data that is resident in caches
- Caches reduce memory access latency when processors accesses data that they have recently accessed! *

* Caches also provide high bandwidth data transfer
The implementation of the linear memory address space abstraction on a modern computer is complex.

The instruction “load the value stored at address X into register R0” might involve a complex sequence of operations by multiple data caches and access to DRAM.

Common organization: hierarchy of caches:
Level 1 (L1), level 2 (L2), level 3 (L3)

Smaller capacity caches near processor → lower latency
Larger capacity caches farther away → larger latency
## Data access times

*(Kaby Lake CPU)*

<table>
<thead>
<tr>
<th>Data in</th>
<th>Latency (number of cycles at 4 GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache</td>
<td>4</td>
</tr>
<tr>
<td>L2 cache</td>
<td>12</td>
</tr>
<tr>
<td>L3 cache</td>
<td>38</td>
</tr>
<tr>
<td>DRAM (best case)</td>
<td>~248</td>
</tr>
</tbody>
</table>
Today

- Today we're talking computer architecture... from a software engineer's perspective

- Key concepts about how modern parallel processors achieve high throughput
  - Two concern parallel execution (multi-core, SIMD parallel execution)
  - One addresses the challenges of memory latency (multi-threading)

- Understanding these basics will help you
  - Understand and optimize the performance of your parallel programs
  - Gain intuition about what workloads might benefit from fast parallel machines
Today's example program

```c
void sinx(int N, int terms, float* x, float* y)
{
    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;
        }
        y[i] = value;
    }
}
```

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

```
x[0]  x[1]  ...  x[N-2]  x[N-1]
```

```
y[0]  y[1]  ...  y[N-2]  y[N-1]
```
void sinx(int N, int terms, float* x, float* y) {
    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;
        }
        y[i] = value;
    }
}
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
```
Superscalar processor

The processor shown here can decode and execute two instructions per clock (if independent instructions exist in an instruction stream)

Note: No ILP exists in this region of the program
Pre multi-core era processor

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

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

Idea #1:

Rather than use transistors to increase sophistication of processor logic that accelerates a single instruction stream (e.g., out-of-order and speculative operations)

Use increasing transistor count to add more cores to the processor
Two cores: compute two elements in parallel

Simpler cores: each core may be 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

This C program will compile to an instruction stream that runs as one thread on one processor core.

If each of the simpler processor cores was 25% slower than the original single complicated one, our program now runs 25% slower than before.

😭
Example: expressing parallelism using C++ threads

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

void my_thread_func(my_args* args)
{
    sinx(args->N, args->terms, args->x, args->y); // do work
}

void parallel_sinx(int N, int terms, float* x, float* y)
{
    std::thread my_thread;
    my_args args;

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

    my_thread = std::thread(my_thread_func, &args); // launch thread
    sinx(N - args.N, terms, x + args.N, y + args.N); // do work on main thread
    my_thread.join(); // wait for thread to complete
}
void sinx(int N, int terms, float* x, float* y) {
    // declares that loop iterations are independent
    forall (int i from 0 to N) {
        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;
        }

        y[i] = value;
    }
}
Four cores: compute four elements in parallel
Sixteen cores: compute sixteen elements in parallel

Sixteen cores, sixteen simultaneous instruction streams
Example: multi-core CPU

Intel “Comet Lake” 10th Generation Core i9 10-core CPU (2020)
Multi-core GPU

GeForce RTX 4090 (2022)
144 processing blocks (called SMs)
Apple A15 Bionic

Two “big cores” + four “small” cores

Image Credit: TechInsights Inc.
Data-parallel expression  (in Kayvon’s fictitious programming language with a “forall” construct)

```c
void sinx(int N, int terms, float* x, float* result)
{
    // declares that loop iterations are independent
    forall (int i from 0 to N)
    {
        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 carry out the exact same sequence of instructions (defined by the loop body), but on different input data given by x[i]

(the loop body computes sine(x[i]))
Add execution units (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
This operation is executed in parallel on all ALUs
Recall our original scalar program

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

Intrinsic datatypes and functions available to C programmers

Intrinsic functions operate on vectors of eight 32-bit values (e.g., vector of 8 floats)
Vector program (using AVX intrinsics)

```c
#include <immintrin.h>

void sinx(int N, int terms, float* x, float* y)
{
    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(&y[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
The program’s use of “forall” declares to the compiler that loop iterations are independent, and that same loop body will be executed on a large number of data elements.

This abstraction can facilitate 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?

```c
forall (int i from 0 to N) {
    float t = x[i];
    <unconditional code>
    if (t > 0.0) {
        t = t * t;
        t = t * 50.0;
        t = t + 100.0;
    } else {
        t = t + 30.0;
        t = t / 10.0;
    }
    <resume unconditional code>
    y[i] = t;
}
```
What about conditional execution?

```plaintext
forall (int i from 0 to N) {
  float t = x[i];
  <unconditional code>
  if (t > 0.0) {
    t = t * t;
    t = t * 50.0;
    t = t + 100.0;
  } else {
    t = t + 30.0;
    t = t / 10.0;
  }
  <resume unconditional code>
  y[i] = t;
}
```
Mask (discard) output of ALU

Not all ALUs do useful work!

Worst case: 1/8 peak performance

forall (int i from 0 to N) {
    float t = x[i];
    <unconditional code>
    if (t > 0.0) {
        t = t * t;
        t = t * 50.0;
        t = t + 100.0;
    } else {
        t = t + 30.0;
        t = t / 10.0;
    }
    <resume unconditional code>
    y[i] = t;
}
After branch: continue at full performance

```java
forall (int i from 0 to N) {
    float t = x[i];
    <unconditional code>
    if (t > 0.0) {
        t = t * t;
        t = t * 50.0;
        t = t + 100.0;
    } else {
        t = t + 30.0;
        t = t / 10.0;
    }
    <resume unconditional code>
    y[i] = t;
}
```
Breakout question

Can you think of piece of code that yields the worst case performance on a processor with 8-wide SIMD execution?

Hint: can you create it using only a single “if” statement?

```c
forall (int i from 0 to N) {
  float t = x[i];
  <unconditional code>
  if (t > 0.0) {
    ???
  } else {
    ???
  }
  <resume unconditional code>
  y[i] = t;
}
```
Some common jargon

- Instruction stream coherence ("coherent execution")
  - Property of a program where the same instruction sequence applies to many data elements
  - Coherent execution IS NECESSARY for SIMD processing resources to be used efficiently
  - Coherent execution IS NOT NECESSARY for efficient parallelization across different cores, since each core has the capability to fetch/decode a different instructions from their thread’s instruction stream

- “Divergent” execution
  - A lack of instruction stream coherence in a program
SIMD execution: modern CPU examples

- Intel AVX2 instructions: 256 bit operations: 8x32 bits or 4x64 bits (8-wide float vectors)
- Intel AVX512 instruction: 512 bit operations: 16x32 bits...
- ARM Neon instructions: 128 bit operations: 4x32 bits...

- 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 by “auto-vectorizing” compiler

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

TL;DR — see “going farther” video

- **“Implicit SIMD”**
  - Compiler generates a binary with scalar instructions
  - But N instances of the program are always run together on the processor
  - Hardware (not compiler) is responsible for simultaneously executing the same instruction from multiple program instances on different data on SIMD ALUs

- **SIMD width of most modern GPUs ranges from 8 to 32**
  - Divergent execution can be a big issue
    (poorly written code might execute at 1/32 the peak capability of the machine!)
Summary: three different forms of parallel execution

- **Superscalar:** exploit ILP within an instruction stream. Process different instructions from the same instruction stream in parallel (within a core)
  - Parallelism automatically discovered by the hardware during execution

- **SIMD:** multiple ALUs controlled by same instruction (within a core)
  - Efficient for data-parallel workloads: amortize control costs over many ALUs
  - Vectorization done by compiler (explicit SIMD) or at runtime by hardware (implicit SIMD)

- **Multi-core:** use multiple processing cores
  - Provides thread-level parallelism: simultaneously execute a completely different instruction stream on each core
  - Software creates threads to expose parallelism to hardware (e.g., via threading API)
My single core, superscalar processor: executes up to two instructions per clock from a single instruction stream (if the instructions are independent).

My dual-core processor: executes one instruction per clock from one instruction stream on each core.

My SIMD quad-core processor: executes one 8-wide SIMD instruction per clock from one instruction stream on each core.
Example: four-core Intel i7-7700K CPU (Kaby Lake)

- 4 core processor
- Three 8-wide SIMD ALUs per core
  (AVX2 instructions)

- 4 cores x 8-wide SIMD x 3 x 4.2 GHz = 400 GFLOPs

* Showing only AVX math units, and fetch/decode unit for AVX (additional capability for integer math)
Example: NVIDIA V100 GPU

80 “SM” cores
128 SIMD ALUs per “SM” (@1.6 GHz) = 16 TFLOPs (~250 Watts)
Part 2: accessing memory
Recall: [very] long latency of data access
(Kaby Lake CPU)

Latency (number of cycles at 4 GHz)

- Data in L1 cache: 4 cycles
- Data in L2 cache: 12 cycles
- Data in L3 cache: 38 cycles
- Data in DRAM (best case): ~248 cycles
Caches reduce length of stalls (reduce memory access latency)

Processors run efficiently when they access data resident in caches
Caches reduce memory access latency when accessing data that they have recently accessed! *

* Caches also provide high bandwidth data transfer
Recall this access pattern

Program reads entire array of 16 bytes, then reads entire array again in the future.

Assume:
Total cache capacity = 8 bytes

Cache has 4-byte cache lines
(So 2 lines fit in cache)

Least recently used (LRU) replacement policy

Discussion Questions:
Why is there no “hit” on second read of address 0x0?
What about second read of address 0x4?
Would your answer change if the cache had a capacity of 4 lines?
Data prefetching reduces stalls *(hides latency)*

- Many modern CPUs have logic for guessing what data will be accessed in the future and “pre-fetching” this data into caches
  - Dynamically analyze program’s memory access patterns to make predictions

- Prefetching 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 *(consumes bandwidth, pollutes caches)*

These loads are cache hits
But what if data hasn’t been read recently, so does not reside in cache?

And the next piece of data to read is not easily predictable?

```c
int x = some_function();
int y = A[x];
```
Consider doing your laundry… Or cooking a meal…

Credit: https://www.theodysseyonline.com/the-dos-and-donts-of-dorm-laundry

Image credit: https://www.escoffier.edu/blog/food-entrepreneurship/culinary-side-hustles/
Multi-threading reduces stalls

- Idea #3: **interleave** processing of multiple threads on the same core to hide stalls
  - If you can’t make progress on the current thread... work on another one
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)

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

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

Time

Runnable

Stall
Hiding stalls with multi-threading

Thread 1
Elements 0 … 7
Runnable
Stall
Runnable
Done!

Thread 2
Elements 8 … 15
Runnable
Stall
Runnable
Done!

Thread 3
Elements 16 … 23
Runnable
Stall
Runnable
Done!

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

Note: during this time, this thread is runnable, but it is not being executed by the processor core.
(The core is executing instructions from another thread.)
No free lunch: storing execution contexts
Consider on-chip storage of execution contexts as a finite resource

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

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

4 hardware threads: storage for large working set per thread
Exercise: consider a simple two-threaded core

Single core processor, multi-threaded core (2 threads). Can run one scalar instruction per clock from one of the hardware threads
What is the utilization of the core? (one thread)

Assume we are running a program where threads perform three arithmetic instructions, followed by memory load (with 12 cycle latency)

\[
\frac{3}{15} = 20\%
\]
What is the utilization of the core? (two threads)

Assume we are running a program where threads perform three arithmetic instructions, followed by memory load (with 12 cycle latency).

\[
\frac{6}{15} = 40\%
\]
How many threads are needed to achieve 100% utilization?

Assume we are running a program where threads perform three arithmetic instructions, followed by memory load (with 12 cycle latency).
Five threads needed to obtain 100% utilization

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4

Five threads required for 100% utilization
Additional threads yield no benefit (already 100% utilization)
How many threads are needed to achieve 100% utilization?

Threads now perform *six arithmetic instructions*, followed by memory load (with 12 cycle latency)
Now only three threads needed for 100% utilization?

Threads now perform *six arithmetic instructions*, followed by memory load (with 12 cycle latency)

100% utilization using only three threads

How does a higher ratio of math instructions to memory latency affect the number of threads needed for latency hiding?
Takeaway (point 1):

A processor with multiple hardware threads has the ability to *avoid stalls* by performing instructions from other threads when one thread must wait for a long latency operation to complete.

Note: the latency of the memory operation is not changed by multi-threading, it just no longer causes reduced processor utilization.
Takeaway (point 2):

A multi-threaded processor hides memory latency by performing arithmetic from other threads.

Programs that feature more arithmetic per memory access need fewer threads to hide memory stalls.
Hardware-supported multi-threading

- Core manages execution contexts for multiple threads
  - 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
  - Processor makes decision about which thread to run each clock

- 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 core’s ALUs

- Simultaneous multi-threading (SMT)
  - Each clock, core chooses instructions from multiple threads to run on ALUs
  - Example: Intel Hyper-threading (2 threads per core)
  - See “going further videos” provided online
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
Two-way multi-threaded cores (2 threads).
Each core can run up to four independent scalar instructions and up to three 8-wide vector instructions (up to 2 vector mul or 3 vector add)

Not shown on this diagram: units for LD/ST operations
NVIDIA V100

- SM = “Streaming Multi-processor”
GPUs: extreme throughput-oriented processors

This is one NVIDIA V100 streaming multi-processor (SM) unit

**Wide SIMD:** 16-wide SIMD ALUs (carry out 32-wide SIMD execute over 2 clocks)

64 x 32 = up to 2048 data items processed concurrently per “SM” core

64 “warp” execution contexts per SM

64 KB registers per sub-core

256 KB registers in total per SM

Registers divided among (up to) 64 “warps” per SM

“Shared” memory + L1 cache storage (128 KB)

- **SIMD fp32 functional unit, control shared across 16 units (16 x MUL-ADD per clock *)**
- **SIMD int functional unit, control shared across 16 units (16 x MUL/ADD per clock **)**
- **SIMD fp64 functional unit, control shared across 8 units (8 x MUL/ADD per clock **)**
- **Tensor core unit**
- **Load/store unit**

* one 32-wide SIMD operation every 2 clocks

** one 32-wide SIMD operation every 4 clocks
NVIDIA V100

There are 80 SM cores on the V100:

That’s 163,840 pieces of data being processed concurrently to get maximal latency hiding!
The story so far…

To utilize modern parallel processors efficiently, an application must:

1. Have sufficient parallel work to utilize all available execution units (across many cores and many execution units per core)

2. Groups of parallel work items must require the same sequences of instructions (to utilize SIMD execution)

3. Expose more parallel work than processor ALUs to enable interleaving of work to hide memory stalls
**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] \times B[i]
- Store result into C[i]

Is this a good application to run on a modern throughput-oriented parallel processor?
To answer this question, we first have to understand the difference between latency and bandwidth

(We’ll start there next time)
Suggestion to students: know these terms

- Instruction stream
- Multi-core processor
- SIMD execution
- Coherent control flow
- Hardware multi-threading
  - Interleaved multi-threading
  - Simultaneous multi-threading
Review

How it all fits together:

- Superscalar execution,
- SIMD execution,
- Multi-core execution,
- And hardware multi-threading

(If you understand this sequence you understand lecture 2)
# Running code on a simple processor

## C program source

```c
void sinx(int N, int terms, float* x, float* y)
{
    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;
        }

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

## Compiled instruction stream (scalar instructions)

```
ld   r0, addr[r1]
mul  r1, r0, r0
add  r2, r0, r0
mul  r3, r1, r2
...
...
...
...
...
st   addr[r2], r0
```
Running code on a simple processor

Instruction stream

```
ld   r0, addr[r1]
mul  r1, r0, r0
add  r2, r0, r0
mul  r3, r1, r2
...  
...  
...  
...  
st   addr[r2], r0
```

Single core processor, single-threaded core. Can run one scalar instruction per clock
Superscalar core

Single core processor, single-threaded core.

Two-way superscalar core: can run up to two independent scalar instructions per clock from one instruction stream (one hardware thread)

Instruction stream

ld  r0, addr[r1]
mul r1, r0, r0
add r2, r0, r0
mul r3, r1, r2
...
...
...
...
...
st  addr[r2], r0

Fetch/Decode
Fetch/Decode
ALU
ALU
Data Cache
Execution Context (HW thread)

Memory

PC
R0
R1
R2
R3
R4
R5
R6
R7
SIMD execution capability

Instruction stream (now with vector instructions)

```
vector_ld  v0, vector_addr[r1]
vector_mul v1, v0, v0
vector_add v2, v0, v0
vector_mul v3, v1, v2
...  
...  
...  
...  
vector_st addr[r2], v0
```

Single core processor, single-threaded core. can run one 8-wide SIMD vector instruction from one instruction stream.
Heterogeneous superscalar (scalar + SIMD)

Instruction stream

```plaintext
vector_ld  v0, vector_addr[r1]
vector_mul v1, v0, v0
add        r2, r1, r0
vector_add v2, v0, v0
vector_mul v3, v1, v2...
...          ...
...          ...
vector_st   addr[r2], v0
```

Single core processor, single-threaded core.
Two-way superscalar core:
can run up to two independent instructions per clock from one instruction stream, provided one is scalar and the other is vector.
Multi-threaded core

Instruction stream 0

```
ld  r0, addr[r1]
mul r1, r0, r0
add r2, r0, r0
mul r3, r1, r2
...
...
...
st  addr[r2], r0
```

Instruction stream 1

```
ld  r0, addr[r1]
sub r1, r0, r0
add r2, r1, r0
mul r5, r1, r0
...
...
...
est addr[r2], r0
```

Note: threads can be running completely different instruction streams (and be at different points in these streams)

Execution of hardware threads is interleaved in time.

Single core processor, multi-threaded core (2 threads). Can run one scalar instruction per clock from one of the instruction streams (hardware threads).
Multi-threaded, superscalar core

Instruction stream 0

- `vector_ld v0, addr[r1]`
- `vector_mul v1, v0, v0`
- `vector_add v2, v1, v0`
- `mul r2, r1, r1`
- ...
- ...
- ...
- `vector_st addr[r2], v0`

Instruction stream 1

- `vector_ld v0, addr[r1]`
- `sub r1, r0, r0`
- `vector_add v2, v0, v0`
- `mul r5, r1, r0`
- ...
- ...
- ...
- `vector_st addr[r2], v0`

Note: threads can be running completely different instruction streams (and be at different points in these streams)

Execution of hardware threads is interleaved in time.

* This detail was an arbitrary decision on this slide: a different implementation of “instruction selection” might run two instructions where one is drawn from each thread, see next slide.
Multi-threaded, superscalar core
(that combines interleaved and simultaneous execution of multiple hardware threads)

Instruction stream 0
vector ld v0, addr[r1]
vectormul v1, v0, v0
mul r2, r1, r1
... ...
... ...
... ...
vecor st addr[r2], v0

Instruction stream 1
vector ld v0, addr[r1]
sub r1, r0, r0
vectoradd v2, v0, v0
mul r5, r1, r0
... ...
... ...
... ...
rect addr[r2], v0

Instruction stream 2
vector ld v0, addr[r1]
vectormul v2, v0, v0
mul r3, r0, r0
sub r1, r0, r3 ...
... ...
... ...
... ...
rect addr[r2], v0

Instruction stream 3
vector ld v0, addr[r1]
sub r3, r0, r0
vectoradd v3, v0, v0
vectoradd v2, v0, v0
mul r2, r1, r1
... ...
... ...
... ...
rect addr[r2], v0

Execution of hardware threads may or may not be interleaved in time
(instructions from different threads may be running simultaneously)
Multi-core, with multi-threaded, superscalar cores

Dual-core processor, multi-threaded cores (4 threads/core).
Two-way superscalar cores: each core can run up to two independent instructions per clock from any of its threads, provided one is scalar and the other is vector.
Example: Intel Skylake/Kaby Lake core

Two-way multi-threaded cores (2 threads). Each core can run up to four independent scalar instructions and up to three 8-wide vector instructions (up to 2 vector mul or 3 vector add)

Not shown on this diagram: units for LD/ST operations
Many modern GPUs execute hardware threads that run instruction streams with only scalar instructions.

GPU cores detect when different hardware threads are executing the same instruction, and implement simultaneous execution of up to SIMD-width threads using SIMD ALUs.

Here ALU 6 would be “masked off” since thread 6 is not executing the same instruction as the other hardware threads.
Thought experiment

- You write a C application that spawns two threads
- 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 the application's threads to the processor’s thread execution contexts?
  Answer: the operating system

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

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