Review

- Flynn Taxonomy of Parallel Architectures
  - SIMD: Single Instruction Multiple Data
  - MIMD: Multiple Instruction Multiple Data
  - SISD: Single Instruction Single Data
  - MISD: Multiple Instruction Single Data (unused)
- Intel SSE SIMD Instructions
  - One instruction fetch that operates on multiple operands simultaneously
  - 64/128 bit XMM registers
  - (SSE = Streaming SIMD Extensions)
- Threads and Thread-level parallelism

Intel SSE Intrinsics

- Intrinsics are C functions and procedures for putting assembly language, including SSE instructions
  - With intrinsics, can program using these instructions indirectly
  - One-to-one correspondence between SSE instructions and intrinsics

Example SSE Intrinsics

- Vector data type: _m128d
- Load and store operations:
  - _mm_load_pd
  - _mm_store_pd
  - _mm_loadu_pd
  - _mm_storeu_pd
- Load and broadcast across vector
  - _mm_load1_pd
- Arithmetic:
  - _mm_add_pd
  - _mm_mul_pd

Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

\[ C_{ij} = (A \times B)_{ij} = \sum_{k=1}^{2} A_{ik} \times B_{kj} \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ C_{11} = 1 \times 0 + 0 \times 1 = 0 \]
\[ C_{12} = 1 \times 0 + 0 \times 1 = 0 \]
\[ C_{21} = 1 \times 1 + 0 \times 2 = 1 \]
\[ C_{22} = 1 \times 1 + 0 \times 1 = 1 \]

Example: 2 x 2 Matrix Multiply

- Using the XMM registers
  - 64-bit/double precision/two doubles per XMM reg

Example:

\[ A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \]
\[ B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \]
\[ C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \]

\[ C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} \]
\[ C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22} \]
\[ C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21} \]
\[ C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22} \]

\[ C_{11} = 1 \times 1 + 0 \times 1 = 1 \]
\[ C_{12} = 1 \times 0 + 0 \times 1 = 0 \]
\[ C_{21} = 0 \times 1 + 1 \times 2 = 2 \]
\[ C_{22} = 0 \times 0 + 1 \times 1 = 1 \]
**Example: 2 x 2 Matrix Multiply**

- **Initialization**

<table>
<thead>
<tr>
<th>C1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **First iteration intermediate result**

<table>
<thead>
<tr>
<th>C1</th>
<th>(A_{1,1}B_{1,1})</th>
<th>(A_{1,2}B_{1,2})</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>(A_{2,1}B_{2,1})</td>
<td>(A_{2,2}B_{2,2})</td>
</tr>
</tbody>
</table>

\[ c1 = _{\text{mm_add}} \cdot _{\text{pd}} [(C1, C2) \cdot (A1, A2); \]  
\[ c2 = _{\text{mm_add}} \cdot _{\text{pd}} [(C1, C2) \cdot (A1, A2); \]  

SSE instructions first do parallel multiplies and then parallel adds in XMM registers

- **I = 1**

A

<table>
<thead>
<tr>
<th>(A_{1,1})</th>
<th>(A_{1,2})</th>
</tr>
</thead>
</table>

\[ _{\text{mm_load}} \cdot _{\text{pd}}: \text{Stored in memory in Column order} \]

B1

<table>
<thead>
<tr>
<th>(B_{1,1})</th>
<th>(B_{1,2})</th>
</tr>
</thead>
</table>

\[ _{\text{mm_load}} \cdot _{\text{pd}}: \text{SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)} \]

B2

<table>
<thead>
<tr>
<th>(B_{2,1})</th>
<th>(B_{2,2})</th>
</tr>
</thead>
</table>

- **Second iteration intermediate result**

<table>
<thead>
<tr>
<th>C1</th>
<th>(A_{1,1}B_{1,1})</th>
<th>(A_{1,2}B_{2,1})</th>
<th>(A_{2,1}B_{1,2})</th>
<th>(A_{2,2}B_{2,2})</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>(A_{1,1}B_{1,1})</td>
<td>(A_{1,2}B_{1,2})</td>
<td>(A_{2,1}B_{1,2})</td>
<td>(A_{2,2}B_{2,2})</td>
</tr>
</tbody>
</table>

\[ c1 = _{\text{mm_add}} \cdot _{\text{pd}} [(C1, C2) \cdot (A1, A2); \]  
\[ c2 = _{\text{mm_add}} \cdot _{\text{pd}} [(C1, C2) \cdot (A1, A2); \]  

SSE instructions first do parallel multiplies and then parallel adds in XMM registers

- **I = 2**

A

<table>
<thead>
<tr>
<th>(A_{1,1})</th>
<th>(A_{1,2})</th>
</tr>
</thead>
</table>

\[ _{\text{mm_load}} \cdot _{\text{pd}}: \text{Stored in memory in Column order} \]

B1

<table>
<thead>
<tr>
<th>(B_{1,1})</th>
<th>(B_{1,2})</th>
</tr>
</thead>
</table>

\[ _{\text{mm_load}} \cdot _{\text{pd}}: \text{SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)} \]

B2

<table>
<thead>
<tr>
<th>(B_{2,1})</th>
<th>(B_{2,2})</th>
</tr>
</thead>
</table>

**Example: 2 x 2 Matrix Multiply**

- **Initialization**

<table>
<thead>
<tr>
<th>C1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **I = 1**

A

<table>
<thead>
<tr>
<th>(A_{1,1})</th>
<th>(A_{1,2})</th>
</tr>
</thead>
</table>

\[ _{\text{mm_load}} \cdot _{\text{pd}}: \text{Load 2 doubles into XMM reg. Stored in memory in Column order} \]

B1

<table>
<thead>
<tr>
<th>(B_{1,1})</th>
<th>(B_{1,2})</th>
</tr>
</thead>
</table>

\[ _{\text{mm_load}} \cdot _{\text{pd}}: \text{SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)} \]

B2

<table>
<thead>
<tr>
<th>(B_{2,1})</th>
<th>(B_{2,2})</th>
</tr>
</thead>
</table>

**Example: 2 x 2 Matrix Multiply**

- **Initialization**

<table>
<thead>
<tr>
<th>C1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **I = 1**

A

<table>
<thead>
<tr>
<th>(A_{1,1})</th>
<th>(A_{1,2})</th>
</tr>
</thead>
</table>

\[ _{\text{mm_load}} \cdot _{\text{pd}}: \text{Load 2 doubles into XMM reg. Stored in memory in Column order} \]

B1

<table>
<thead>
<tr>
<th>(B_{1,1})</th>
<th>(B_{1,2})</th>
</tr>
</thead>
</table>

\[ _{\text{mm_load}} \cdot _{\text{pd}}: \text{SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)} \]

B2

<table>
<thead>
<tr>
<th>(B_{2,1})</th>
<th>(B_{2,2})</th>
</tr>
</thead>
</table>

**Example: 2 x 2 Matrix Multiply**

- **Initialization**

<table>
<thead>
<tr>
<th>C1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **I = 1**

A

<table>
<thead>
<tr>
<th>(A_{1,1})</th>
<th>(A_{1,2})</th>
</tr>
</thead>
</table>

\[ _{\text{mm_add}} \cdot _{\text{pd}}: \text{Stored in memory in Column order} \]

B1

<table>
<thead>
<tr>
<th>(B_{1,1})</th>
<th>(B_{1,2})</th>
</tr>
</thead>
</table>

\[ _{\text{mm_add}} \cdot _{\text{pd}}: \text{SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)} \]

B2

<table>
<thead>
<tr>
<th>(B_{2,1})</th>
<th>(B_{2,2})</th>
</tr>
</thead>
</table>

**Example: 2 x 2 Matrix Multiply**

- **Initialization**

<table>
<thead>
<tr>
<th>C1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **I = 1**

A

<table>
<thead>
<tr>
<th>(A_{1,1})</th>
<th>(A_{1,2})</th>
</tr>
</thead>
</table>

\[ _{\text{mm_load}} \cdot _{\text{pd}}: \text{Stored in memory in Column order} \]

B1

<table>
<thead>
<tr>
<th>(B_{1,1})</th>
<th>(B_{1,2})</th>
</tr>
</thead>
</table>

\[ _{\text{mm_load}} \cdot _{\text{pd}}: \text{SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)} \]

B2

<table>
<thead>
<tr>
<th>(B_{2,1})</th>
<th>(B_{2,2})</th>
</tr>
</thead>
</table>

**Example: 2 x 2 Matrix Multiply**

**Definition of Matrix Multiply:**

\[ C_{ij} = (A \times B)_{ij} = \sum_{k=1}^{2} A_{ik} \times B_{kj} \]

\[
\begin{array}{ccc}
A_{i1} & A_{i2} \\
B_{11} & B_{12} & B_{12} \\
1 & 3 & 2 \\
0 & 1 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22} \\
\end{bmatrix}
\end{array} =
\begin{array}{c}
\begin{bmatrix}
A_{i1} & A_{i2} \\
B_{11} & B_{12} & B_{12} \\
1 & 3 & 2 \\
0 & 1 & 4 \\
\end{bmatrix}
\end{array}
\]

Garcia, Spring 2014 © UCB

Garcia, Spring 2014 © UCB
Example: 2 x 2 Matrix Multiply (Part 1 of 2)

```
L2: movapd (%rax,%rsi), %xmm1
    // Load aligned A[i,j+1] -> m1
    movddup (%rdx), %xmm0
    // Load B[i], duplicate -> m0
    mulpd %xmm1, %xmm0
    // Multiply m0 * m1 -> m0
    addpd %xmm0, %xmm3
    // Add m0 + m3 -> m3
    movddup (%rdx), %xmm0
    // Load B[i+1], duplicate -> m0
    mulpd %xmm0, %xmm1
    // Multiply m0 * m1 -> m1
    addpd %xmm1, %xmm2
    // Add m1 + m2 -> m2
    addq %rsi, %rsi
    // rax = rax + 1
    addq %rax, %rdx
    cmpq $32, %rax
    jne L2
    // jump to L2 if not equal
    movapd %xmm3, (%rcx)
    // store aligned m3 into C[l,k+1]
    movapd %xmm2, (%rdi)
    // store aligned m2 into C[l,j+1]
```

Example: 2 x 2 Matrix Multiply (Part 2 of 2)

```
for i = 0 to 2
for j = 0 to 2
    c00 = a00*b00 + a01*b01 + a02*b02
    c01 = a00*b10 + a01*b11 + a02*b12
    c02 = a00*b20 + a01*b21 + a02*b22
    c10 = a10*b00 + a11*b01 + a12*b02
    c11 = a10*b10 + a11*b11 + a12*b12
    c12 = a10*b20 + a11*b21 + a12*b22
    c20 = a20*b00 + a21*b01 + a22*b02
    c21 = a20*b10 + a21*b11 + a22*b12
    c22 = a20*b20 + a21*b21 + a22*b22
end for
store aligned m2 into C[l,l+1]
jump to L2 if not equal
```

Thoughts about Threads

“Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly non-deterministic, and the job of the programmer becomes one of pruning that nondeterminism.”
— The Problem with Threads, Edward A. Lee, UC Berkeley, 2006

Background: Threads

- A **Thread** stands for “thread of execution”, is a single stream of instructions
  - A program / process can split, or fork itself into separate threads, which can (in theory) execute simultaneously.
  - An easy way to describe/think about parallelism
- A single CPU can execute many threads by **Time Division Multiplexing**
- **Multithreading** is running multiple threads through the same hardware
Parallel Processing: Multithreaded Systems (MIMD)

- Multithreaded (MIMD): a computer system with at least 2 processors
- New term: multithreading
- Multithreaded, Microprocessor, MIMD

1. Deliver high throughput for independent jobs via thread-level parallelism
2. Improve the run time of a single program that has been specially crafted to run on a multiprocessor - a parallel processing program

Now Use term core for processor ("Multicore") because "Multithreaded Microprocessor" is too redundant

Clicker Question

What significant thing happened in computer architecture around 2005?

a) CPU heat densities approached nuclear reactors
b) They started slowing the clock speeds down
c) Power drain of CPUs hit a plateau
d) CPU single-core performance hit a plateau
e) CPU manufacturers started offering only multi-core CPUs for desktops and laptops

Parallel Performance Over Time

<table>
<thead>
<tr>
<th>Year</th>
<th>Cores</th>
<th>SIMD bits/Core</th>
<th>Core × SIMD bits</th>
<th>Peak DP FLOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>2003</td>
<td>2</td>
<td>128</td>
<td>256</td>
<td>4</td>
</tr>
<tr>
<td>2005</td>
<td>4</td>
<td>128</td>
<td>512</td>
<td>8</td>
</tr>
<tr>
<td>2007</td>
<td>6</td>
<td>128</td>
<td>768</td>
<td>12</td>
</tr>
<tr>
<td>2009</td>
<td>8</td>
<td>128</td>
<td>1024</td>
<td>16</td>
</tr>
<tr>
<td>2011</td>
<td>10</td>
<td>256</td>
<td>2560</td>
<td>40</td>
</tr>
<tr>
<td>2013</td>
<td>12</td>
<td>256</td>
<td>3072</td>
<td>48</td>
</tr>
<tr>
<td>2015</td>
<td>14</td>
<td>512</td>
<td>7168</td>
<td>112</td>
</tr>
<tr>
<td>2017</td>
<td>16</td>
<td>512</td>
<td>8192</td>
<td>128</td>
</tr>
<tr>
<td>2019</td>
<td>18</td>
<td>1024</td>
<td>18432</td>
<td>288</td>
</tr>
<tr>
<td>2021</td>
<td>20</td>
<td>1024</td>
<td>20480</td>
<td>320</td>
</tr>
</tbody>
</table>

- Only path to performance is parallelism
  - Clock rates flat or declining
  - SIMD: 2X width every 3-4 years
    - 128b wide now, 256b in 2013, 512b in 20147, 1024b in 2018?
  - Advanced Vector Extensions are 256-bits wide!
  - MIMD: Add 2 cores every 2 years: 2, 4, 6, 8, 10, ...
- A key challenge is to craft parallel programs that have high performance on multiprocessors as the number of processors increase – i.e., that scale
- Scheduling, load balancing, time for synchronization, overhead for communication
- Will explore this further in labs and projects

So, In Conclusion...

- Sequential software is slow software
  - SIMD and MIMD only path to higher performance
- SSE Intrinsics allow SIMD instructions to be invoked from C programs
- MIMD uses multithreading to achieve high parallelism