Lecture 20
Almost Thread Level Parallelism

Da’Miki

Intel Xeon Phi

Click to learn more

intel.com/xeonphi
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 in 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

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

Corresponding SSE instructions:

- MOVAPD/aligned, packed double
- MOVAPD/aligned, packed double
- MOVUPD/unaligned, packed double
- MOVUPD/unaligned, packed double
- MOVSD + shuffling/duplicating
- ADDPD/add, packed double
- MULPD/multiple, packed double
Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

\[ C_{i,j} = (A \times B)_{i,j} = \sum_{k=1}^{2} A_{i,k} \times B_{k,j} \]

\[
\begin{bmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2}
\end{bmatrix}
\times
\begin{bmatrix}
B_{1,1} & B_{1,2} \\
B_{2,1} & B_{2,2}
\end{bmatrix}
= \begin{bmatrix}
C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\
C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}
\end{bmatrix}
\]

\[
= \begin{bmatrix}
C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\
C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\times
\begin{bmatrix}
1 & 3 \\
2 & 4
\end{bmatrix}
= \begin{bmatrix}
C_{1,1} = 1*1 + 0*2 = 1 \\
C_{2,1} = 0*1 + 1*2 = 2
\end{bmatrix}
\]

\[
= \begin{bmatrix}
C_{1,2} = 1*3 + 0*4 = 3 \\
C_{2,2} = 0*3 + 1*4 = 4
\end{bmatrix}
\]
Example: 2 x 2 Matrix Multiply

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

\[
\begin{array}{c|c}
C_1 & C_{1,1} & C_{2,1} \\
\hline
C_2 & C_{1,2} & C_{2,2} \\
\end{array}
\]

Stored in memory in Column order

\[
\begin{array}{c|c}
A \quad & A_{1,i} & A_{2,i} \\
\end{array}
\]

\[
\begin{array}{c|c}
B_1 & B_{i,1} & B_{i,1} \\
B_2 & B_{i,2} & B_{i,2} \\
\end{array}
\]
Example: 2 x 2 Matrix Multiply

• Initialization

<table>
<thead>
<tr>
<th></th>
<th>(A_{1,1})</th>
<th>(A_{2,1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C_1)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(C_2)</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\_mm_load_pd: Stored in memory in Column order
\_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register.
Example: 2 x 2 Matrix Multiply

• Initialization

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

• \( I = 1 \)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>_mm_load_pd: Load 2 doubles into XMM reg, Stored in memory in Column order</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁₁</td>
<td>A₂₁</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>B₁</th>
<th>_mm_load1_pd: 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)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B₁₁</td>
<td>B₁₂</td>
<td></td>
</tr>
</tbody>
</table>

Garcia, Lustig Fall 2014 © UCB
Example: 2 x 2 Matrix Multiply

• First iteration intermediate result

<table>
<thead>
<tr>
<th>C₁</th>
<th>0+A₁,₁B₁,₁</th>
<th>0+A₂,₁B₁,₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>C₂</td>
<td>0+A₁,₁B₁,₂</td>
<td>0+A₂,₁B₁,₂</td>
</tr>
</tbody>
</table>

c₁ = _mm_add_pd(c1, _mm_mul_pd(a, b1));
c₂ = _mm_add_pd(c2, _mm_mul_pd(a, b2));
SSE instructions first do parallel multiplies and then parallel adds in XMM registers

• l = 1

<table>
<thead>
<tr>
<th>A</th>
<th>A₁,₁</th>
<th>A₂,₁</th>
</tr>
</thead>
</table>

_mm_load_pd: Stored in memory in Column order

<table>
<thead>
<tr>
<th>B₁</th>
<th>B₁,₁</th>
<th>B₁,₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>B₂</td>
<td>B₁,₂</td>
<td>B₁,₂</td>
</tr>
</tbody>
</table>

_mm_load1_pd: 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)
Example: 2 x 2 Matrix Multiply

• First iteration intermediate result

<table>
<thead>
<tr>
<th>C1</th>
<th>0+A_{1,1}B_{1,1}</th>
<th>0+A_{2,1}B_{1,1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>0+A_{1,1}B_{1,2}</td>
<td>0+A_{2,1}B_{1,2}</td>
</tr>
</tbody>
</table>

\[ c1 = \texttt{\_mm\_add\_pd}(c1, \texttt{\_mm\_mul\_pd}(a, b1)); \]
\[ c2 = \texttt{\_mm\_add\_pd}(c2, \texttt{\_mm\_mul\_pd}(a, b2)); \]
SSE instructions first do parallel multiplies and then parallel adds in XMM registers

• \( I = 2 \)

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

\texttt{\_mm\_load\_pd}: Stored in memory in Column order

<table>
<thead>
<tr>
<th>B_1</th>
<th>B_{2,1}</th>
<th>B_{2,1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_2</td>
<td>B_{2,2}</td>
<td>B_{2,2}</td>
</tr>
</tbody>
</table>

\texttt{\_mm\_load1\_pd}: 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)
Example: 2 x 2 Matrix Multiply

- Second iteration intermediate result

\[
\begin{array}{c|c|c}
C_1 & C_1,1 & C_2,1 \\
\hline
C_1 & A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\
C_2 & A_{1,1}B_{1,2} + A_{1,2}B_{2,2} & A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \\
\end{array}
\]

\[
c_1 = _\text{mm_add_pd}(c_1, _\text{mm_mul_pd}(a, b_1));
\]

\[
c_2 = _\text{mm_add_pd}(c_2, _\text{mm_mul_pd}(a, b_2));
\]

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

- \( I = 2 \)

\[
\begin{array}{c|c|c}
A & A_{1,2} & A_{2,2} \\
\hline
B_1 & B_{2,1} & B_{2,1} \\
B_2 & B_{2,2} & B_{2,2} \\
\end{array}
\]

_\text{mm_load_pd}: Stored in memory in Column order

_\text{mm_load1_pd}: 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)
Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

\[ C_{i,j} = (A \times B)_{i,j} = \sum_{k=1}^{2} A_{i,k} \times B_{k,j} \]

\[
\begin{bmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2}
\end{bmatrix}
\times
\begin{bmatrix}
B_{1,1} & B_{1,2} \\
B_{2,1} & B_{2,2}
\end{bmatrix}
= 
\begin{bmatrix}
C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\
C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}
\end{bmatrix}
\times
\begin{bmatrix}
C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\
C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\times
\begin{bmatrix}
1 & 3 \\
2 & 4
\end{bmatrix}
= 
\begin{bmatrix}
C_{1,1} = 1*1 + 0*2 = 1 \\
C_{2,1} = 0*1 + 1*2 = 2
\end{bmatrix}
\times
\begin{bmatrix}
C_{1,2} = 1*3 + 0*4 = 3 \\
C_{2,2} = 0*3 + 1*4 = 4
\end{bmatrix}
\]
Example: 2 x 2 Matrix Multiply
(Part 1 of 2)

```
#include <stdio.h>
// header file for SSE compiler intrinsics
#include <emmintrin.h>

#include <stdio.h>  // header file for SSE compiler intrinsics
#include <emmintrin.h>

// NOTE: vector registers will be represented in comments as v1 = [a | b]
// where v1 is a variable of type __m128d and a, b are doubles

int main(void) {
    int lda = 2;
    int i = 0;
    // declare several 128-bit vector variables

    // allocate A,B,C aligned on 16-byte boundaries
    double B[4] __attribute__((aligned (16)));
    double C[4] __attribute__((aligned (16)));

    // initialize A, B, C for example
    /* A = 1 0 */
    /* B = 1 3 */
    B[0] = 1.0; B[1] = 2.0; B[2] = 3.0; B[3] = 4.0;
    /* C = 0 0 */
    C[0] = 0.0; C[1] = 0.0; C[2] = 0.0; C[3] = 0.0;
```
Example: 2 x 2 Matrix Multiply
(Part 2 of 2)

// used aligned loads to set
// c1 = [c_11 | c_21]
c1 = _mm_load_pd(C+0*lda);
// c2 = [c_12 | c_22]
c2 = _mm_load_pd(C+1*lda);

for (i = 0; i < 2; i++) {
    /* a =
       i = 0: [a_11 | a_21]
       i = 1: [a_12 | a_22]
    */
a = _mm_load_pd(A+i*lda);
    /* b1 =
       i = 0: [b_11 | b_11]
       i = 1: [b_21 | b_21]
    */
b1 = _mm_load1_pd(B+i+0*lda);
    /* b2 =
       i = 0: [b_12 | b_12]
       i = 1: [b_22 | b_22]
    */
b2 = _mm_load1_pd(B+i+1*lda);

    /* c1 =
       i = 0: [c_11 + a_11*b_11 | c_21 + a_21*b_11]
       i = 1: [c_11 + a_21*b_21 | c_21 + a_22*b_21]
    */
c1 = _mm_add_pd(c1, _mm_mul_pd(a, b1));
    /* c2 =
       i = 0: [c_12 + a_11*b_12 | c_22 + a_21*b_12]
       i = 1: [c_12 + a_21*b_22 | c_22 + a_22*b_22]
    */
c2 = _mm_add_pd(c2, _mm_mul_pd(a, b2));
}

// store c1,c2 back into C for completion
_mm_store_pd(C+0*lda,c1);
_mm_store_pd(C+1*lda,c2);

// print C
printf("%g,%g\n%g,%g\n",C[0],C[2],C[1],C[3]);
return 0;
}
L2: movapd (%rax,%rsi), %xmm1 //Load aligned A[i,i+1]->m1
movddup (%rdx), %xmm0 //Load B[j], duplicate->m0
mulpd %xmm1, %xmm0 //Multiply m0*m1->m0
addpd %xmm0, %xmm3 //Add m0+m3->m3
movddup 16(%rdx), %xmm0 //Load B[j+1], duplicate->m0
mulpd %xmm0, %xmm1 //Multiply m0*m1->m1
addpd %xmm1, %xmm2 //Add m1+m2->m2
addq $16, %rax // rax+16 -> rax (i+=2)
addq $8, %rdx // rdx+8 -> rdx (j+=1)
cmpq $32, %rax // rax == 32?
jne L2 // jump to L2 if not equal
movapd %xmm3, (%rcx) //store aligned m3 into C[k,k+1]
movapd %xmm2, (%rdi) //store aligned m2 into C[l,l+1]
You Are Here!

**Software**

- Parallel Requests
  Assigned to computer
  e.g., Search “Katz”

- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads

- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions

- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words

- Hardware descriptions
  All gates functioning in parallel at same time

---

**Hardware**

- Warehouse Scale Computer
  Harness Parallelism & Achieve High Performance

- Computer

  - Core
  - ... Core

  - Memory
    - (Cache)

  - Input/Output

- Instruction Unit(s)
  - $A_0 + B_0$, $A_1 + B_1$, $A_2 + B_2$, $A_3 + B_3$

- Functional Unit(s)

- Main Memory

- Logic Gates
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*

  ![Diagram showing CPU and threads](image)

  - Blue: Thread $0$
  - Red: Thread $1$
  - Purple: Thread $2$

• *Multithreading* is running multiple threads through the same hardware
Parallel Processing: Multiprocessor Systems (MIMD)

- **Multiprocessor (MIMD):** a computer system with at least 2 processors

1. Deliver high throughput for independent jobs via job-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 “Multiprocessor Microprocessor” too redundant
Transition to Multicore

Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond
Multiprocessors and You

• Only path to performance is parallelism
  – Clock rates flat or declining
  – SIMD: 2X width every 3-4 years
    • 128b wide now, 256b 2011, 512b in 2014?, 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
## 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>
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