Thread Level Parallelism

1997: THE FIRST INTEL® TERAFLOP COMPUTER consisted of:
9,298 INTEL PROCESSORS and occupied: 72 SERVER CABINETS

THE INTEL® XEON® PHI™ COPROCESSOR will provide: 1 TERAFLOP OF PERFORMANCE and occupy: 1 PCIe SLOT

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:

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

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

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

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

\[
\begin{array}{cc}
C_1 & C_{1,1} & C_{2,1} \\
C_2 & C_{1,2} & C_{2,2} \\
A & A_{1,i} & A_{2,i} \\
B_1 & B_{i,1} & B_{i,1} \\
B_2 & B_{i,2} & B_{i,2} \\
\end{array}
\]

Stored in memory in Column order
Example: 2 x 2 Matrix Multiply

• Initialization

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_1 )</td>
<td>0</td>
</tr>
<tr>
<td>( C_2 )</td>
<td>0</td>
</tr>
</tbody>
</table>
Example: 2 x 2 Matrix Multiply

• Initialization

<table>
<thead>
<tr>
<th></th>
<th>C_1</th>
<th></th>
<th>C_2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

• \( I = 1 \)

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

_\text{\_mm\_load\_pd}: Load 2 doubles into XMM reg, 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

• 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>

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

• \( l = 1 \)

A

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

_mm_load_pd: Stored in memory in Column order

B1

<table>
<thead>
<tr>
<th>B1,1</th>
<th>B1,1</th>
</tr>
</thead>
</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></th>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0+A_{1,1}B_{1,1}</td>
<td>0+A_{2,1}B_{1,1}</td>
</tr>
<tr>
<td></td>
<td>0+A_{1,1}B_{1,2}</td>
<td>0+A_{2,1}B_{1,2}</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

- I = 2

A

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

_B1

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

_B2

<table>
<thead>
<tr>
<th></th>
<th>B_{2,2}</th>
<th>B_{2,2}</th>
</tr>
</thead>
</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 (duplicates value in both halves of XMM)
Example: 2 x 2 Matrix Multiply

- Second iteration intermediate result

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

- \( I = 2 \)

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

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

\[
c_1 = \_mm\_add\_pd(c1, \_mm\_mul\_pd(a,b1));
\]

\[
c_2 = \_mm\_add\_pd(c2, \_mm\_mul\_pd(a,b2));
\]

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

\[
\_mm\_load\_pd: \text{Stored in memory in Column order}
\]

\[
\_mm\_load1\_pd: \text{SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)}
\]
Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

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

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

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

```c
#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) {
  // allocate A,B,C aligned on 16-byte boundaries
  double B[4] __attribute__((aligned (16)));
  double C[4] __attribute__((aligned (16)));
  int lda = 2;
  int i = 0;
  // declare several 128-bit vector variables
  __m128d c1,c2,a,b1,b2;
  // Initialize A, B, C for example
  /* A =
     1 0
     0 1
   */
  /* B =
     1 3
     2 4
   */
  B[0] = 1.0; B[1] = 2.0; B[2] = 3.0; B[3] = 4.0;
  /* C =
     0 0
     0 0
   */
  C[0] = 0.0; C[1] = 0.0; C[2] = 0.0; C[3] = 0.0;
```

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

// used aligned loads to set
// c1 = \[c_{11} \mid c_{21}\]
c1 = _mm_load_pd(C+0*lda);
// c2 = \[c_{12} \mid c_{22}\]
c2 = _mm_load_pd(C+1*lda);

for (i = 0; i < 2; i++) {
    /* a = */
    i = 0: \[a_{11} \mid a_{21}\]
    i = 1: \[a_{12} \mid a_{22}\]
    */
    a = _mm_load_pd(A+i*lda);
    /* b1 = */
    i = 0: \[b_{11} \mid b_{11}\]
    i = 1: \[b_{21} \mid b_{21}\]
    */
    b1 = _mm_load1_pd(B+i+0*lda);
    /* b2 = */
    i = 0: \[b_{12} \mid b_{12}\]
    i = 1: \[b_{22} \mid b_{22}\]
    */
    b2 = _mm_load1_pd(B+i+1*lda);

    /* c1 = */
    i = 0: \[c_{11} + a_{11}*b_{11} \mid c_{21} + a_{21}*b_{11}\]
    i = 1: \[c_{11} + a_{21}*b_{21} \mid c_{21} + a_{22}*b_{21}\]
    */
    c1 = _mm_add_pd(c1,_mm_mul_pd(a,b1));
    /* c2 = */
    i = 0: \[c_{12} + a_{11}*b_{12} \mid c_{22} + a_{21}*b_{12}\]
    i = 1: \[c_{12} + a_{21}*b_{22} \mid c_{22} + a_{22}*b_{22}\]
    */
    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;
Inner loop from gcc –O -S

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

Harnes
Parallelism &
Achieve High
Performance

- Hardware descriptions
  All gates functioning in parallel at same time

Smart Phone

Warehouse Scale Computer

Computer

Main Memory

Intruction Unit(s)

Functional Unit(s)

A0+B0, A1+B1, A2+B2, A3+B3

Logic Gates

Input/Output

Project 3

Garcia, Spring 2014 © UCB
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: 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
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
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