CS 61C: Great Ideas in Computer Architecture

Flynn Taxonomy, Data-level Parallelism

Instructors:
Vladimir Stojanovic & Nicholas Weaver

http://inst.eecs.berkeley.edu/~cs61c/
New-School Machine Structures (It’s a bit more complicated!)

- **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 @ one time

- **Programming Languages**

**Software**

**Hardware**

Harness
Parallelism & Achieve High Performance

- **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
Using Parallelism for Performance

• Two basic ways:
  – Multiprogramming
    • run multiple independent programs in parallel
    • “Easy”
  – Parallel computing
    • run one program faster
    • “Hard”

• We’ll focus on parallel computing for next few lectures
Single-Instruction/Single-Data Stream (SISD)

- Sequential computer that exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are traditional uniprocessor machines
Single-Instruction/Multiple-Data Stream (SIMD or “sim-dee”)

- SIMD computer exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., Intel SIMD instruction extensions or NVIDIA Graphics Processing Unit (GPU)
Multiple-Instruction/Multiple-Data Streams (MIMD or “mim-dee”)

- Multiple autonomous processors simultaneously executing different instructions on different data.
  - MIMD architectures include multicore and Warehouse-Scale Computers
Multiple-Instruction/Single-Data Stream (MISD)

- Multiple-Instruction, Single-Data stream computer that exploits multiple instruction streams against a single data stream.
  - Rare, mainly of historical interest only
Flynn* Taxonomy, 1966

<table>
<thead>
<tr>
<th>Instruction Streams</th>
<th>Single</th>
<th>Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>SISD: Intel Pentium 4</td>
<td>SIMD: SSE instructions of x86</td>
</tr>
<tr>
<td>Multiple</td>
<td>MISD: No examples today</td>
<td>MIMD: Intel Xeon e5345 (Clovertown)</td>
</tr>
</tbody>
</table>

- In 2013, SIMD and MIMD most common parallelism in architectures – usually both in same system!
- Most common parallel processing programming style: Single Program Multiple Data (“SPMD”)
  - Single program that runs on all processors of a MIMD
  - Cross-processor execution coordination using synchronization primitives
- SIMD (aka hw-level *data parallelism*): specialized function units, for handling lock-step calculations involving arrays
  - Scientific computing, signal processing, multimedia (audio/video processing)

*Prof. Michael Flynn, Stanford*
SIMD Architectures

• *Data parallelism*: executing same operation on multiple data streams

• Example to provide context:
  – Multiplying a coefficient vector by a data vector (e.g., in filtering)
    \[ y[i] := c[i] \times x[i], \ 0 \leq i < n \]

• Sources of performance improvement:
  – One instruction is fetched & decoded for entire operation
  – Multiplications are known to be independent
  – Pipelining/concurrency in memory access as well
Intel “Advanced Digital Media Boost”

- To improve performance, Intel’s SIMD instructions
  - Fetch one instruction, do the work of multiple instructions
First SIMD Extensions: MIT Lincoln Labs TX-2, 1957
Intel SIMD Extensions

• MMX 64-bit registers, reusing floating-point registers [1992]
• SSE2/3/4, new 128-bit registers [1999]
• AVX, new 256-bit registers [2011]
  – Space for expansion to 1024-bit registers
• AVX-512 [2013]
### XMM Registers

<table>
<thead>
<tr>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMM7</td>
</tr>
<tr>
<td>XMM6</td>
</tr>
<tr>
<td>XMM5</td>
</tr>
<tr>
<td>XMM4</td>
</tr>
<tr>
<td>XMM3</td>
</tr>
<tr>
<td>XMM2</td>
</tr>
<tr>
<td>XMM1</td>
</tr>
<tr>
<td>XMM0</td>
</tr>
</tbody>
</table>

- Architecture extended with eight 128-bit data registers: XMM registers
  - x86 64-bit address architecture adds 8 additional registers (XMM8 – XMM15)
Intel Architecture SSE2+
128-Bit SIMD Data Types

- Note: in Intel Architecture (unlike MIPS) a word is 16 bits
  - Single-precision FP: Double word (32 bits)
  - Double-precision FP: Quad word (64 bits)
## SSE/SSE2 Floating Point Instructions

<table>
<thead>
<tr>
<th>Data transfer</th>
<th>Arithmetic</th>
<th>Compare</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm</td>
<td>ADD{SS/PS/SD/PD} xmm, mem/xmm</td>
<td>CMP{SS/PS/SD/PD}</td>
</tr>
<tr>
<td>SUB{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV {H/L} {PS/PD} xmm, mem/xmm</td>
<td>MUL{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DIV{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SQRT{SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAX {SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIN{SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
</tbody>
</table>

- xmm: one operand is a 128-bit SSE2 register
- mem/xmm: other operand is in memory or an SSE2 register
- {SS} Scalar Single precision FP: one 32-bit operand in a 128-bit register
- {PS} Packed Single precision FP: four 32-bit operands in a 128-bit register
- {SD} Scalar Double precision FP: one 64-bit operand in a 128-bit register
- {PD} Packed Double precision FP, or two 64-bit operands in a 128-bit register
- {A} 128-bit operand is aligned in memory
- {U} means the 128-bit operand is unaligned in memory
- {H} means move the high half of the 128-bit operand
- {L} means move the low half of the 128-bit operand
Packed and Scalar Double-Precision Floating-Point Operations

Packed

Scalar
Example: SIMD Array Processing

for each $f$ in array
    $f = \sqrt{f}$

for each $f$ in array
{
    load $f$ to the floating-point register
    calculate the square root
    write the result from the register to memory
}

for each 4 members in array
{
    load 4 members to the SSE register
    calculate 4 square roots in one operation
    store the 4 results from the register to memory
}

SIMD style
Data-Level Parallelism and SIMD

• SIMD wants adjacent values in memory that can be operated in parallel
• Usually specified in programs as loops
  \[
  \text{for}(i=1000; \ i>0; \ i=i-1) \\
  \quad x[i] = x[i] + s;
  \]
• How can reveal more data-level parallelism than available in a single iteration of a loop?
• *Unroll loop* and adjust iteration rate
Looping in MIPS

Assumptions:
- $t1$ is initially the address of the element in the array with the highest address
- $f0$ contains the scalar value $s$
- $8(t2)$ is the address of the last element to operate on

CODE:
Loop: 1. l.d $f2,0(t1)$ ; $f2$=array element
   2. add.d $f10,$f2,$f0 ; add s to $f2$
   3. s.d $f10,0(t1)$ ; store result
   4. addiu $t1,$t1,#-8 ; decrement pointer 8 byte
   5. bne $t1,$t2,Loop ;repeat loop if $t1 != $t2
Loop Unrolled

Loop:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>l.d</td>
<td>$f2,0($t1)</td>
</tr>
<tr>
<td>add.d</td>
<td>$f10,$f2,$f0</td>
</tr>
<tr>
<td>s.d</td>
<td>$f10,0($t1)</td>
</tr>
<tr>
<td>l.d</td>
<td>$f4,-8($t1)</td>
</tr>
<tr>
<td>add.d</td>
<td>$f12,$f4,$f0</td>
</tr>
<tr>
<td>s.d</td>
<td>$f12,-8($t1)</td>
</tr>
<tr>
<td>l.d</td>
<td>$f6,-16($t1)</td>
</tr>
<tr>
<td>add.d</td>
<td>$f14,$f6,$f0</td>
</tr>
<tr>
<td>s.d</td>
<td>$f14,-16($t1)</td>
</tr>
<tr>
<td>l.d</td>
<td>$f8,-24($t1)</td>
</tr>
<tr>
<td>add.d</td>
<td>$f16,$f8,$f0</td>
</tr>
<tr>
<td>s.d</td>
<td>$f16,-24($t1)</td>
</tr>
<tr>
<td>addiu</td>
<td>$t1,$t1,#-32</td>
</tr>
<tr>
<td>bne</td>
<td>$t1,$t2,Loop</td>
</tr>
</tbody>
</table>

NOTE:
1. Only 1 Loop Overhead every 4 iterations
2. This unrolling works if 
   \[ \text{loop\_limit}(\text{mod} \ 4) = 0 \]
3. Using different registers for each iteration eliminates data hazards in pipeline
### Loop Unrolled Scheduled

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>l.d</code></td>
<td>Load $f2,0($t1)</td>
</tr>
<tr>
<td><code>l.d</code></td>
<td>Load $f4,8($t1)</td>
</tr>
<tr>
<td><code>l.d</code></td>
<td>Load $f6,16($t1)</td>
</tr>
<tr>
<td><code>l.d</code></td>
<td>Load $f8,24($t1)</td>
</tr>
<tr>
<td><code>add.d</code></td>
<td>Add $f10,$f2,$f0</td>
</tr>
<tr>
<td><code>add.d</code></td>
<td>Add $f12,$f4,$f0</td>
</tr>
<tr>
<td><code>add.d</code></td>
<td>Add $f14,$f6,$f0</td>
</tr>
<tr>
<td><code>add.d</code></td>
<td>Add $f16,$f8,$f0</td>
</tr>
<tr>
<td><code>s.d</code></td>
<td>Store $f10,0($t1)</td>
</tr>
<tr>
<td><code>s.d</code></td>
<td>Store $f12,16($t1)</td>
</tr>
<tr>
<td><code>s.d</code></td>
<td>Store $f14,24($t1)</td>
</tr>
<tr>
<td><code>addiu</code></td>
<td>Addi $t1,$t1,#-32</td>
</tr>
<tr>
<td><code>bne</code></td>
<td>Branch on Not Equal $t1,$t2,Loop</td>
</tr>
</tbody>
</table>

- **4 Loads side-by-side**: Could replace with 4-wide SIMD Load
- **4 Adds side-by-side**: Could replace with 4-wide SIMD Add
- **4 Stores side-by-side**: Could replace with 4-wide SIMD Store
Loop Unrolling in C

• Instead of compiler doing loop unrolling, could do it yourself in C

```c
for(i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

• Could be rewritten

```c
for(i=1000; i>0; i=i-4) {
    x[i] = x[i] + s;
    x[i-1] = x[i-1] + s;
    x[i-2] = x[i-2] + s;
    x[i-3] = x[i-3] + s;
}
```

What is downside of doing it in C?
Generalizing Loop Unrolling

• A loop of \textbf{n iterations}

• \textbf{k copies} of the body of the loop

• \textbf{Assuming (n mod k) \neq 0}

Then we will run the loop with 1 copy of the body \textbf{(n mod k)} times and with \textbf{k copies of the body floor(n/k)} times
Example: Add Two Single-Precision Floating-Point Vectors

Computation to be performed:

```c
vec_res.x = v1.x + v2.x;
vec_res.y = v1.y + v2.y;
vec_res.z = v1.z + v2.z;
vec_res.w = v1.w + v2.w;
```

SSE Instruction Sequence:

(Note: Destination on the right in x86 assembly)
```
movaps address-of-v1, %xmm0
  // v1.w | v1.y | v1.x -> xmm0
addps address-of-v2, %xmm0
  // v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x -> xmm0
movaps %xmm0, address-of-vec_res
```

mov a ps: move from mem to XMM register, memory aligned, packed single precision
add ps: add from mem to XMM register, packed single precision
mov a ps: move from XMM register to mem, memory aligned, packed single precision
Administrativia

• MT2 is Monday, April 4th, 7-9pm:
  – Covers lecture material up till and including 3/28 lecture (Amdahl’s law)
  – Conflict: Email Fred or William by midnight today
  – Watch Piazza for locations announcement

• Guerrilla Session: Floating Point & Performance
  – Wed 3/30 3 - 5 PM @ 241 Cory
  – Sat 4/02 1 - 3 PM @ 521 Cory

• Project 3-2 feedback
Intel SSE Intrinsics

• Intrinsics are C functions and procedures for inserting assembly language into C code, including SSE instructions
  – With intrinsics, can program using these instructions indirectly
  – One-to-one correspondence between SSE instructions and intrinsics
Example SSE Intrinsics

**Instrinsics:**

- **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:

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

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

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

\[ \begin{bmatrix}
1 \\
2
\end{bmatrix} \]
Example: 2 x 2 Matrix Multiply

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

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

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

Stored in memory in Column order

\[
\begin{bmatrix}
C_{1,1} & C_{1,2} \\
C_{2,1} & C_{2,2}
\end{bmatrix}
\]

\[
\begin{bmatrix}
C_1 & C_2
\end{bmatrix}
\]
Example: 2 x 2 Matrix Multiply

• Initialization

\[
\begin{array}{c|c}
C_1 & 0 & 0 \\
\hline
C_2 & 0 & 0 \\
\end{array}
\]
Example: 2 x 2 Matrix Multiply

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

- **Initialization**

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

- **i = 1**

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

_\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

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

- **First iteration intermediate result**

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

\(c_1 = \_mm\_add\_pd(c_1, \_mm\_mul\_pd(a, b_1))\);
\(c_2 = \_mm\_add\_pd(c_2, \_mm\_mul\_pd(a, b_2))\);

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

- **i = 1**

\(\_mm\_load\_pd\): 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

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

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

- First iteration intermediate result

\[
\begin{array}{c|c}
C_1 & 0+A_{1,1}B_{1,1} & 0+A_{2,1}B_{1,1} \\
\hline
C_2 & 0+A_{1,1}B_{1,2} & 0+A_{2,1}B_{1,2}
\end{array}
\]

\[
c_1 = \_\text{mm\_add\_pd}(c1, \_\text{mm\_mul\_pd}(a,b1));
\]

\[
c_2 = \_\text{mm\_add\_pd}(c2, \_\text{mm\_mul\_pd}(a,b2));
\]

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

- **Second iteration intermediate result**

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

- **i = 2**

  \[
  c_1 = \_mm\_add\_pd(c_1, \_mm\_mul\_pd(a, b_1));
  c_2 = \_mm\_add\_pd(c_2, \_mm\_mul\_pd(a, b_2));
  \]

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

  - `_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
(Part 1 of 2)

```
#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) {
  double B[4] __attribute__((aligned (16)));
  double C[4] __attribute__((aligned (16)));
  int lda = 2;
  int i = 0;

  // 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;
}
```
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*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;
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]
And in Conclusion, ...

- Flynn Taxonomy
- Intel SSE SIMD Instructions
  - Exploit data-level parallelism in loops
  - One instruction fetch that operates on multiple operands simultaneously
  - 128-bit XMM registers
- SSE Instructions in C
  - Embed the SSE machine instructions directly into C programs through use of intrinsics
  - Achieve efficiency beyond that of optimizing compiler