CS61C
Great Ideas in Computer Architecture (a.k.a. Machine Structures)

Parallelism

UC Berkeley
Teaching Professor
Dan Garcia

UC Berkeley
Professor
Bora Nikolić

Berkeley
UNIVERSITY OF CALIFORNIA

cs61c.org
New-School Machine Structures

Software
Parallel Requests
Assigned to computer
  e.g., Search “Cats”

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 work in parallel at same time

Harness Parallelism & Achieve High Performance

Hardware

Computer
Core
Memory (Cache)
Input/Output
Exec. Unit(s)
Functional Block(s)

Logic Gates

Smart Phone
Warehouse
Scale
Computer

Harness Parallelism & Achieve High Performance

Parallelism (2)
Application: Machine Learning

- Inference in machine learning applications

Matrix-vector multiplications

<table>
<thead>
<tr>
<th>Layer</th>
<th>Weights</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>W0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1</td>
<td>W1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>W2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ln-1</td>
<td>Wn-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

"cat"
Matrix multiplication
- Basic operation in many engineering, data, and imaging processing tasks
- Image filtering, noise reduction, machine learning...
- Many closely related operations

*dgemm*
- double-precision floating-point matrix multiplication
  - In FORTRAN
Square matrix of dimension NxN
Matrix Multiplication

\[
C = A \times B \\
C_{ij} = \Sigma_k (A_{ik} \times B_{kj})
\]
Matrix Multiplication
### Example: 2 x 2 Matrix Multiply

Matrix Multiply:

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

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

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

```python
def dgemm(N, a, b, c):
    for i in range(N):
        for j in range(N):
            c[i+j*N] = 0
            for k in range(N):
                c[i+j*N] += a[i+k*N] * b[k+j*N]
```

<table>
<thead>
<tr>
<th>N</th>
<th>Python [MFLOPs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>5.4</td>
</tr>
<tr>
<td>160</td>
<td>5.5</td>
</tr>
<tr>
<td>480</td>
<td>5.4</td>
</tr>
<tr>
<td>960</td>
<td>5.3</td>
</tr>
</tbody>
</table>

- 1 MFLOP = 1 Million floating-point operations per second (fadd, fmul)
- dgemm(N ...) takes $2N^3$ FLOPs
- $c = a \times b$

- $a$, $b$, $c$ are $N \times N$ matrices

```c
void dgemm_scalar(int N, double *a, double *b, double *c) {
    for (int i=0; i<N; i++)
        for (int j=0; j<N; j++)
            double cij = 0;
            for (int k=0; k<N; k++)
                // $a[i][k] \times b[k][j]$
                cij += a[i+k*N] * b[k+j*N];
                // $c[i][j]$
                c[i+j*N] = cij;
}
```
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void) {
    // start time
    // Note: clock() measures execution time, not real time
    // big difference in shared computer environments
    // and with heavy system load
    clock_t start = clock();

    // task to time goes here:
    // dgemm(N, ...);

    // "stop" the timer
    clock_t end = clock();

    // compute execution time in seconds
    double delta_time = (double)(end-start)/CLOCKS_PER_SEC;
}
```
Which class gives you this kind of power? We could stop here ... but why? Let’s do better!

<table>
<thead>
<tr>
<th>N</th>
<th>C [GFLOPS]</th>
<th>Python [GFLOPS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1.30</td>
<td>0.0054</td>
</tr>
<tr>
<td>160</td>
<td>1.30</td>
<td>0.0055</td>
</tr>
<tr>
<td>480</td>
<td>1.32</td>
<td>0.0054</td>
</tr>
<tr>
<td>960</td>
<td>0.91</td>
<td>0.0053</td>
</tr>
</tbody>
</table>

Garcia, Nikolić
Flynn’s Taxonomy
### Software vs. Hardware Parallelism

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>Matrix Multiply written in MatLab running on an Intel Pentium 4</td>
<td>Windows Vista Operating System running on an Intel Pentium 4</td>
</tr>
<tr>
<td>Parallel</td>
<td>Matrix Multiply written in MATLAB running on an Intel Core i7</td>
<td>Windows Vista Operating System running on an Intel Core i7</td>
</tr>
</tbody>
</table>

- Choice of hardware and software parallelism are independent
  - Concurrent software can also run on serial hardware
  - Sequential software can also run on parallel hardware

- **Flynn’s Taxonomy** is for parallel hardware
SIMD and MIMD most commonly encountered today

Most common parallel processing programming style: Single Program Multiple Data (“SPMD”)
- Single program that runs on all processors of an MIMD
- Cross-processor execution coordination through conditional expressions (will see later in Thread Level Parallelism)

SIMD: specialized function units (hardware), for handling lock-step calculations involving arrays
- Scientific computing, machine learning, signal processing, multimedia (audio/video processing)
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

PU = Processing Unit

This is what we did up to now in 61C

Parallelism (16)
Computer that applies a single instruction stream to multiple data streams for operations that may be naturally parallelized (e.g. SIMD instruction extensions or Graphics Processing Unit)

This segment

Parallelism (17)
Multiple Instruction/Multiple Data Stream (MIMD)

- Multiple autonomous processors simultaneously executing different instructions on different data.
- MIMD architectures include multicore and Warehouse Scale Computers.

Later in this module: Parallelism (18)
Exploits multiple instruction streams against a single data stream for data operations that can be naturally parallelized (e.g. certain kinds of array processors)

MISD no longer commonly encountered, mainly of historical interest only

This has few applications. Not covered in 61C.
SIMD Architectures
Parallelism (21)

Garcia, Nikolić

Data-Level Parallelism (DLP):
operation on multiple data streams

Example: Multiplying a coefficient vector by a data vector (e.g. in filtering)

\[ y[i] := c[i] \times x[i], \quad 0 \leq i < n \]

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
First SIMD Extensions: MIT Lincoln Labs TX-2, 1957

<table>
<thead>
<tr>
<th>Description</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONE 36 BIT AE (36)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>OPERAND WORD STRUCTURE</td>
<td></td>
</tr>
<tr>
<td>TWO 18 BIT AE’S (18,18)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>OPERAND WORD STRUCTURE</td>
<td></td>
</tr>
<tr>
<td>ONE 27 BIT AE &amp; ONE 9 BIT AE (27,9)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>OPERAND WORD STRUCTURE</td>
<td></td>
</tr>
<tr>
<td>FOUR 9 BIT AE’S (6,9,9,9)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>OPERAND WORD STRUCTURE</td>
<td></td>
</tr>
</tbody>
</table>

Garcia, Nikolić
To improve performance, Intel’s SIMD instructions
- Fetch one instruction, do the work of multiple instructions
- MMX (MultiMedia eXtension, Pentium II processor family)
- **SSE (Streaming SIMD Extension, Pentium III and beyond)**
Parallelism

• Started with multimedia extensions (MMX)
• New instructions every few years
• New and wider registers
• More parallelism
C vs. Python

<table>
<thead>
<tr>
<th>N</th>
<th>AVX [GFLOPS]</th>
<th>C [GFLOPS]</th>
<th>Python [GFLOPS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>4.56</td>
<td>1.30</td>
<td>0.0054</td>
</tr>
<tr>
<td>160</td>
<td>5.47</td>
<td>1.30</td>
<td>0.0055</td>
</tr>
<tr>
<td>480</td>
<td>5.27</td>
<td>1.32</td>
<td>0.0054</td>
</tr>
<tr>
<td>960</td>
<td>3.64</td>
<td>0.91</td>
<td>0.0053</td>
</tr>
</tbody>
</table>

Theoretical Intel i7-5557U performance is ~25 GFLOPS

3.1GHz x 2 instructions/cycle x 4 mults/inst = 24.8GFLOPS

Garcia, Nikolić
XMM Registers in SSE

- Architecture extended with eight 128-bit data registers
  - 64-bit address architecture: available as 16 64-bit registers (XMM8 – XMM15)
  - e.g. 128-bit packed single-precision floating-point data type (doublewords), allows four single-precision operations to be performed simultaneously
- Note: in Intel Architecture (unlike RISC V) a word is 16 bits
  - Single precision FP: Double word (32 bits)
  - Double precision FP: Quad word (64 bits)
Parallelism

High amounts of compute need large amounts of state to compensate for memory BW. AVX512 has 8x state compared to SSE (commensurate with its 8x flops level)

Intel confidential — presented under NDA only — under embargo until 6:01 a.m. PDT, June 19, 2017
SIMD Array Processing
Example: SIMD Array Processing

```python
for each f in array:
    f = sqrt(f)
```

```python
def process_array(array):
    for f in array:
        load f to the floating-point register
        calculate the square root
        write the result from the register to memory
```

```python
for every 4 members in array {
    load 4 members to the SSE register
    calculate 4 square roots in one operation
    write the result from the register to memory
}
```
Example: Add Single-Precision FP Vectors

Computation to be performed:

\[
\begin{align*}
vec\_res.x &= v1.x + v2.x; \\
vec\_res.y &= v1.y + v2.y; \\
vec\_res.z &= v1.z + v2.z; \\
vec\_res.w &= v1.w + v2.w;
\end{align*}
\]

SSE Instruction Sequence:

\[
\begin{align*}
\text{movaps} & \quad \text{address-of-v1, } \%\text{xmm0} \\
// & \quad v1.w \mid v1.z \mid v1.y \mid v1.x \rightarrow \%\text{xmm0} \\
\text{addps} & \quad \text{address-of-v2, } \%\text{xmm0} \\
// & \quad v1.w+v2.w \mid v1.z+v2.z \mid v1.y+v2.y \mid v1.x+v2.x \\
& \quad \rightarrow \%\text{xmm0} \\
\text{movaps} & \quad \%\text{xmm0}, \text{address-of-vec\_res}
\end{align*}
\]

move from mem to XMM register
memory aligned, packed single precision
add from mem to XMM register
packed single precision
move from XMM register to mem
memory aligned, packed single precision
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

**Intel SSE Intrinsics**

**Intrinsics:**
- Vector data type: `_m128d`
- Load and store operations:
  - `_mm_load_pd` MOVAPD/aligned, packed double
  - `_mm_store_pd` MOVAPD/aligned, packed double
- Arithmetic:
  - `_mm_add_pd` ADDPD/add, packed double
  - `_mm_mul_pd` MULPD/multiple, packed double

**Corresponding SSE instructions:**
- MOVAPD/aligned, packed double
- MOVAPD/aligned, packed double
- ADDPD/add, packed double
- MULPD/multiple, packed double
Matrix Multiply Example
Example: 2 x 2 Matrix Multiply

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

- Initialization

<table>
<thead>
<tr>
<th></th>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example: 2 x 2 Matrix Multiply

- **Initialization**

  - $I = 1$

  ![Matrix Multiplication Diagram]

  - $A_{1,1}$, $A_{1,2}$, $B_{1,1}$, $B_{1,2}$, $C_{1,1}$, $C_{1,2}$, $C_{2,1}$, $C_{2,2}$

  - $\_\_mm\_load\_pd$: 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

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

- \( I = 1 \), intermediate result

- \( c_1 = \_mm\_add\_pd(c_1, \_mm\_mul\_pd(a, b1)) \)
- \( c_2 = \_mm\_add\_pd(c_2, \_mm\_mul\_pd(a, b2)) \)
SSE instructions first do parallel multiplies and then parallel adds in XMM registers

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

- \_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

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

- **I = 2, intermediate result**

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

\[c_1 = \_mm\_add\_pd(c1, \_mm\_mul\_pd(a, b1));\]
\[c_2 = \_mm\_add\_pd(c2, \_mm\_mul\_pd(a, b2));\]

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

- **\(_{mm\_load\_pd}\):** 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

- I = 2, intermediate result

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

\[
c1 = \_mm_add_pd(c1, \_mm_mul_pd(a, b1));
\]

\[
c2 = \_mm_add_pd(c2, \_mm_mul_pd(a, b2));
\]

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

- **_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
#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 =                           (note column order!)
    1 0
    0 1 */
    /* B =                              (note column order!)
    1 3
    2 4 */
    B[0] = 1.0;  B[1] = 2.0;  B[2] = 3.0;  B[3] = 4.0;
    /* C =                             (note column order!)
    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
Example: 2 x 2 Matrix Multiply

// 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
%g,%g",C[0],C[2],C[1],C[3]);
return 0;
/* 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));
}
To improve RISC-V performance, add SIMD instructions (and hardware) – V extension

- Fetch one instruction, do the work of multiple instructions
- OP denotes a vector instruction, prefix v – vector register
- \texttt{vadd \ vd, \ vs1, \ vs2} (adds two vectors stored in vector registers)
- Assume vectors are 512-bits wide
“And in Conclusion…”

- 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 AVX SIMD Instructions
  - One instruction fetch that operates on multiple operands simultaneously
  - 512/256/128/64-bit AVX registers
  - Use C intrinsics