Parallelism
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

Warehouse Scale Computer

Smart Phone

Computer

Core

Memory

(Input/Output)

Exec. Unit(s)

Functional Block(s)

Main Memory

Logic Gates

A0+B0

A1+B1

Out = A0+B0 + A1+B1
Inference in machine learning applications

Matrix-vector multiplications
Reference Problem: Matrix Multiplication

- 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

\[ C = A \times B \]

\[ C_{ij} = \sum_k (A_{ik} \times B_{kj}) \]
Matrix Multiplication
Example: 2 x 2 Matrix Multiply

Matrix Multiply:

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

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

\[
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 3 \\
2 & 4
\end{bmatrix}
= 
\begin{bmatrix}
C_{1,1} = 1*1 + 0*2 = 1 & C_{1,2} = 1*3 + 0*4 = 3 \\
C_{2,1} = 0*1 + 1*2 = 2 & C_{2,2} = 0*3 + 1*4 = 4
\end{bmatrix}
\]
## 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)
- \( \text{dgemm}(N \ldots) \) takes \( 2*N^3 \) FLOPs
- $c = a \times b$
- $a, b, c$ are $N \times N$ matrices

```c
// Scalar; P&H p. 226
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] * 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

This is what we did up to now in 61C
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)
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
Multiple Instruction/Single Data Stream (MISD)

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

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

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

**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>ONE 36 BIT AE</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(36)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPERAND WORD</td>
<td>S</td>
<td>35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STRUCTURE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| TWO 18 BIT AE'S | 4 | 3 | 2 | 1 |
| (18,18)        |     |     |     |     |
| OPERAND WORD   | S  | 17 | S  | 17 |
| STRUCTURE      |     |     |     |     |

| ONE 27 BIT & D | 4 | 3 | 2 | 1 |
| ONE 9 BIT AE C |     |     |     |     |
| (27,9)         |     |     |     |     |
| OPERAND WORD   | S  | 26 | S  | 8 |
| STRUCTURE      |     |     |     |     |

| FOUR 9 BIT AE'S | 4 | 3 | 2 | 1 |
| (9,9,9,9)       |     |     |     |     |
| OPERAND WORD    | S  | S  | S  | S |
| STRUCTURE       |     |     |     |     |
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)
Intel x86 SIMD Evolution

- Started with multimedia extensions (MMX)
- New instructions every few years
- New and wider registers
- More parallelism

Cornea, ARITH-22, 2015
**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
- 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

<table>
<thead>
<tr>
<th>XMM7</th>
<th>XMM6</th>
<th>XMM5</th>
<th>XMM4</th>
<th>XMM3</th>
<th>XMM2</th>
<th>XMM1</th>
<th>XMM0</th>
</tr>
</thead>
</table>
- 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)
SIMD Registers in AVX512

AVX512 state

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
Check Out My Laptop (lscpu)

<table>
<thead>
<tr>
<th>Model:</th>
<th>126</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model name:</td>
<td>Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz</td>
</tr>
<tr>
<td>Stepping:</td>
<td>5</td>
</tr>
<tr>
<td>CPU MHz:</td>
<td>1497.605</td>
</tr>
<tr>
<td>BogoMIPS:</td>
<td>2995.21</td>
</tr>
<tr>
<td>Hypervisor vendor:</td>
<td>Microsoft</td>
</tr>
<tr>
<td>Virtualization type:</td>
<td>full</td>
</tr>
<tr>
<td>L1d cache:</td>
<td>192 KiB</td>
</tr>
<tr>
<td>L1i cache:</td>
<td>128 KiB</td>
</tr>
<tr>
<td>L2 cache:</td>
<td>2 MiB</td>
</tr>
<tr>
<td>L3 cache:</td>
<td>8 MiB</td>
</tr>
<tr>
<td>Vulnerability Itlb multihit:</td>
<td>KVM: Vulnerable</td>
</tr>
<tr>
<td>Vulnerability L1tf:</td>
<td>Not affected</td>
</tr>
<tr>
<td>Vulnerability Mds:</td>
<td>Not affected</td>
</tr>
<tr>
<td>Vulnerability Meltdown:</td>
<td>Not affected</td>
</tr>
<tr>
<td>Vulnerability Spec store bypass v1:</td>
<td>Mitigation; Speculative Store Bypass disabled via prctl and seccomp</td>
</tr>
<tr>
<td>Vulnerability Spectre v1:</td>
<td>Mitigation; usercopy/swappgs barriers and _-_user pointer sanitization</td>
</tr>
<tr>
<td>Vulnerability Spectre v2:</td>
<td>Mitigation; Enhanced IBRS, IBPB conditional, ASB filling</td>
</tr>
</tbody>
</table>

Flags:
- rmu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr pdpe1gb k8 pdmovb pdmovq pmrd pmwr pmi cmov sse urpide cx16 mhz aperf pti cmov pse36 cmovs cmovb intel_pмяt ste pdmpe36 sse2 tsc dvd cmova cmovb pdmpe36 tpxe
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 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:

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

SSE Instruction Sequence:

\begin{align*}
\text{movaps address-of-v1, %xmm0} & \quad \text{move from mem to XMM register} \\
& \quad \text{memory aligned, packed single precision} \\
\text{addps address-of-v2, %xmm0} & \quad \text{add from mem to XMM register} \\
& \quad \text{packed single precision} \\
\text{movaps %xmm0, address-of-vec\_res} & \quad \text{move from XMM register to mem} \\
& \quad \text{memory aligned, packed single precision}
\end{align*}
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

**Intrinsics:**
- Vector data type: 
  \_m128d
- Load and store operations:
  \_mm\_load\_pd
  \_mm\_store\_pd
- Arithmetic:
  \_mm\_add\_pd
  \_mm\_mul\_pd

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

Matrix Multiply:

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

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

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

Garcia, Nikolić
Example: 2 x 2 Matrix Multiply

- Initialization

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

- **Initialization**

  \[
  \begin{array}{cc}
  C_1 & \begin{array}{c}
  0 \\
  0
  \end{array} \\
  C_2 & \begin{array}{c}
  0 \\
  0
  \end{array}
  \end{array}
  \]

- **I = 1**

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

_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

- I = 1, intermediate result

\[
\begin{align*}
C_1 & \quad 0 + A_{1,1}B_{1,1} & 0 + A_{2,1}B_{1,1} \\
C_2 & \quad 0 + A_{1,1}B_{1,2} & 0 + A_{2,1}B_{1,2}
\end{align*}
\]

\[
c_1 = \texttt{_mm_add_pd}(c_1, \texttt{_mm_mul_pd}(a,b1))
\]

\[
c_2 = \texttt{_mm_add_pd}(c_2, \texttt{_mm_mul_pd}(a,b2))
\]

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

\[
\texttt{_mm_load_pd}: \text{ Stored in memory in Column order}
\]

\[
\texttt{_mm_load1_pd}: \text{ SSE instruction that loads a double word and stores it in the high and low double words of the XMM register}
\]

Garcia, Nikolić
Example: 2 x 2 Matrix Multiply

\[ C_1 \begin{bmatrix} 0 + A_{1,1}B_{1,1} & 0 + A_{2,1}B_{1,1} \\ 0 + A_{1,1}B_{1,2} & 0 + A_{2,1}B_{1,2} \end{bmatrix} \]

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

---

I = 2, intermediate result

A \begin{bmatrix} A_{1,2} & A_{2,2} \end{bmatrix}

_mm_load_pd: Stored in memory in Column order

B_1 \begin{bmatrix} B_{2,1} & B_{2,1} \end{bmatrix}

_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

\[
\begin{align*}
C_1 &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} + A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\
C_2 &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} + A_{2,1}B_{1,2} + A_{2,2}B_{2,2}
\end{align*}
\]

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

- **_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

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

SSE instructions first do parallel multiplies and then parallel adds in XMM registers.
Example: 2 x 2 Matrix Multiply

```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;
    int c = 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

// 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_12]
       i = 1: [b_21 | b_22]
    */
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;
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