61C Survey

It would be nice to have a review lecture every once in a while, actually showing us how things fit in the bigger picture
Agenda

• 61C – the big picture
• Parallel processing
• Single instruction, multiple data
• SIMD matrix multiplication
• Loop unrolling
• Memory access strategy - blocking
• And in Conclusion, ...
61C Topics so far ...

• What we learned:
  1. Binary numbers
  2. C
  3. Pointers
  4. Assembly language
  5. Processor micro-architecture
  6. Pipelining
  7. Caches
  8. Floating point

• What does this buy us?
  – Promise: execution speed
  – Let’s check!
Reference Problem

• **Matrix multiplication**
  – Basic operation in many engineering, data, and imaging processing tasks
    – Ex.: Image filtering, noise reduction, ...
  – Core operation in Neural Nets and Deep Learning
    – Image classification (cats ...)
    – Robot Cars
    – Machine translation
    – Fingerprint verification
    – Automatic game playing

• **dgemm**
  – double-precision floating-point general matrix-multiply
  – Standard well-studied and widely used routine
  – Part of Linpack/Lapack
2D-Matrices

• Square matrix of dimension NxN
  • $i$ indexes through rows
  • $j$ indexes through columns
Matrix Multiplication

\[ C = A \times B \]

\[ C_{ij} = \sum_k (A_{ik} \times B_{kj}) \]
2D Matrix Memory Layout

- $a_{ij}$ in C uses row-major
- Fortran uses column-major
- Our examples use column-major

$a_{ij}: a[i*N + j]$  
$a_{ij}: a[i + j*N]$
dgemm Reference Code: Python

• Linear addressing, assumes “column-major” memory layout

```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 $2 \times N^3$ flops
\[ \mathbf{c} = \mathbf{a} \times \mathbf{b} \]

- \( \mathbf{c}, \mathbf{a}, \mathbf{b} \) are \( N \times N \) matrices

C

```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++)
                // \( \mathbf{a}[i][k] \times \mathbf{b}[k][j] \)
                cij += a[i+k*N] * b[k+j*N];
            // \( \mathbf{c}[i][j] \)
            c[i+j*N] = cij;
}
```

Lecture 18: Parallel Processing - SIMD
#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;
}

C versus Python

<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>

Which other class gives you this kind of power? We could stop here ... but why? Let’s do better!

240x!
Agenda

• 61C – the big picture
• Parallel processing
• Single instruction, multiple data
• SIMD matrix multiplication
• Amdahl’s law
• Loop unrolling
• Memory access strategy - blocking
• And in Conclusion, ...
Why Parallel Processing?

• CPU Clock Rates are no longer increasing
  – Technical & economic challenges
    ▪ Advanced cooling technology too expensive or impractical for most applications
    ▪ Energy costs are prohibitive

• Parallel processing is only path to higher speed
  – Compare airlines:
    ▪ Maximum air-speed limited by economics
    ▪ Use more and larger airplanes to increase throughput
    ▪ (And smaller seats ...)

CS 61c
Using Parallelism for Performance

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

• We’ll focus on parallel computing in the next few lectures
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

Harvest

Parallelism & Achieve High Perform

Hardware

Warehouse Scale Computer

Today’s Lecture

A₀ + B₀, A₁ + B₁, A₂ + B₂, A₃ + B₃

Logic Gates
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
  - E.g. Our RISC-V processor

This is what we did up to now in 61C
Single-Instruction/Multiple-Data Stream (SIMD or “sim-dee”)

- SIMD computer processes multiple data streams using a single instruction stream, e.g., Intel SIMD instruction extensions or NVIDIA Graphics Processing Unit (GPU)

Today’s topic.
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

Topic of Lecture 19 and beyond.
Multiple-Instruction/Single-Data Stream (MISD)

- Multiple-Instruction, Single-Data stream computer that processes multiple instruction streams with a single data stream.
  - Historical significance

This has few applications. Not covered in 61C.
• SIMD and MIMD are currently the 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

*Prof. Michael Flynn, Stanford
NEW YORK, NY, March 21, 2018 – ACM, the Association for Computing Machinery, today named John L. Hennessy, former President of Stanford University, and David A. Patterson, retired Professor of the University of California, Berkeley, recipients of the 2017 ACM A.M. Turing Award for pioneering a systematic, quantitative approach to the design and evaluation of computer architectures with enduring impact on the microprocessor industry. Hennessy and Patterson created a systematic and quantitative approach to designing faster, lower power, and reduced instruction set computer (RISC) microprocessors.
Agenda

• 61C – the big picture
• Parallel processing
• Single instruction, multiple data
• SIMD matrix multiplication
• Amdahl’s law
• Loop unrolling
• Memory access strategy - blocking
• And in Conclusion, ...
SIMD – “Single Instruction Multiple Data”
SIMD (Vector) Mode
SIMD Applications & Implementations

• Applications
  – Scientific computing
    ▪ Matlab, NumPy
  – Graphics and video processing
    ▪ Photoshop, ...
  – Big Data
    ▪ Deep learning
  – Gaming
  – ...

• Implementations
  – x86
  – ARM
  – RISC-V vector extensions
First SIMD Extensions: MIT Lincoln Labs TX-2, 1957
Intel x86: SIMD: Continuous Evolution

MMX 1997

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE</td>
<td>SSE2</td>
<td>SSE3</td>
<td>SSSE3</td>
<td>SSE4.1</td>
<td>SSE4.2</td>
<td>AES-NI</td>
<td>AVX</td>
</tr>
</tbody>
</table>

- **1999**: SSE
  - 70 instructions
  - Single-Precision Vectors
  - Streaming operations

- **2000**: SSE2
  - 144 instructions
  - Double-precision Vectors
  - 8/16/32 integer
  - 64/128-bit vector

- **2004**: SSE3
  - 13 instructions
  - Complex Data

- **2006**: SSSE3
  - 32 instructions
  - Decode

- **2007**: SSE4.1
  - 47 instructions
  - Video
  - Graphics building blocks
  - Advanced vector instr

- **2008**: SSE4.2
  - 8 instructions
  - String/XML processing
  - POP-Count
  - CRC

- **2009**: AES-NI
  - 7 instructions
  - Encryption and Decryption
  - Key Generation

- **2010/11**: AVX
  - ~100 new instructions
  - ~300 legacy instructions
  - Updated to 256-bit vector
  - 3 and 4-operand instructions
Intel Advanced Vector eXtensions

<table>
<thead>
<tr>
<th>Year</th>
<th>GFLOPS</th>
<th>Processor</th>
<th>Technology</th>
<th>SIMD Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>87</td>
<td>Westmere</td>
<td>32 nm</td>
<td>SSE 4.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DDR3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PCIe2</td>
</tr>
<tr>
<td>2012</td>
<td>185</td>
<td>Sandy Bridge</td>
<td>32 nm</td>
<td>AVX (256 bit registers)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DDR3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PCIe3</td>
</tr>
<tr>
<td>2013</td>
<td>~225</td>
<td>Ivy Bridge</td>
<td>22 nm</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>~500</td>
<td>Haswell</td>
<td>22 nm</td>
<td>AVX2 (new instructions)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DDR4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PCIe3</td>
</tr>
<tr>
<td>2015</td>
<td>tbd</td>
<td>Broadwell</td>
<td>14 nm</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Future</td>
<td>tbd</td>
<td>Skylake</td>
<td>14 nm</td>
<td>AVX 3.2 (512 bit registers)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DDR4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PCIe4</td>
</tr>
</tbody>
</table>

AVX also supported by AMD processors

AVX Registers getting wider, instruction set getting richer

Laptop CPU Specs

$ sysctl -a | grep cpu
hw.physicalcpu: 2
hw.logicalcpu: 4

machdep.cpu.brand_string:
   Intel(R) Core(TM) i7-5557U CPU @ 3.10GHz

machdep.cpu.features: FPU VME DE PSE TSC MSR PAE MCE CX8 APIC SEP MTRR PGE MCA CMOV PAT PSE36 CLFSH DS ACPI MMX FXSR SSE SSE2 SS HTT TM PBE SSE3 PCLMULQDQ DTES64 MON DSCPL VMX EST TM2 SSSE3 FMA CX16 TPR PDCM SSE4.1 SSE4.2 x2APIC MOVBE POPCNT AES PCID XSAVE OSXSAVE SEGLIM64 TSCTMR AVX1.0 RDRAND F16C

machdep.cpu.leaf7_features: SMEP ERMS RDWRFSGS TSC_THREAD_OFFSET BMI1 AVX2 BMI2_INVPCID SMAP RDSEED ADX IPT FPU_CSDS
AVX SIMD Registers
SIMD Data Types

(Now also AVX-512 available)
Agenda

• 61C – the big picture
• Parallel processing
• Single instruction, multiple data
• SIMD matrix multiplication
• Amdahl’s law
• Loop unrolling
• Memory access strategy - blocking
• And in Conclusion, ...
Problem

• Today’s compilers can generate SIMD code
• But in some cases, better results by hand (assembly)
• We will study x86 (not using RISC-V as no vector hardware widely available yet)
  – Over 1000 instructions to learn ...
• Can we use the compiler to generate all non-SIMD instructions?
x86 SIMD “Intrinsics”


4 parallel multiplies

2 instructions per clock cycle (CPI = 0.5)

Intrinsic assembly instruction
Intrinsics: Direct access to assembly from C

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>__m256</td>
<td>256-bit as eight single-precision floating-point values, representing a YMM register or memory location</td>
</tr>
<tr>
<td>__m256d</td>
<td>256-bit as four double-precision floating-point values, representing a YMM register or memory location</td>
</tr>
<tr>
<td>__m256i</td>
<td>256-bit as integers, (bytes, words, etc.)</td>
</tr>
<tr>
<td>__m128</td>
<td>128-bit single precision floating-point (32 bits each)</td>
</tr>
<tr>
<td>__m128d</td>
<td>128-bit double precision floating-point (64 bits each)</td>
</tr>
</tbody>
</table>
Intrinsics AVX Code Nomenclature

<table>
<thead>
<tr>
<th>Marking</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[s/d]</td>
<td>Single- or double-precision floating point</td>
</tr>
<tr>
<td>[i/u]nnn</td>
<td>Signed or unsigned integer of bit size $nnn$, where $nnn$ is 128, 64, 32, 16, or 8</td>
</tr>
<tr>
<td>[ps/pd/sd]</td>
<td>Packed single, packed double, or scalar double</td>
</tr>
<tr>
<td>epi32</td>
<td>Extended packed 32-bit signed integer</td>
</tr>
<tr>
<td>si256</td>
<td>Scalar 256-bit integer</td>
</tr>
</tbody>
</table>
## Raw Double-Precision Throughput

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>i7-5557U</td>
</tr>
<tr>
<td>Clock rate (sustained)</td>
<td>3.1 GHz</td>
</tr>
<tr>
<td>Instructions per clock (mul_pd)</td>
<td>2</td>
</tr>
<tr>
<td>Parallel multiplies per instruction</td>
<td>4</td>
</tr>
<tr>
<td>Peak double FLOPS</td>
<td>24.8 GFLOPS</td>
</tr>
</tbody>
</table>

Actual performance is lower because of overhead

https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/
for i ...; i+=4

for j ...

**Inner Loop:**

```c
__m256d c0 = {0,0,0,0};
for (int k=0; k<N; k++) {
    c0 = _mm256_fmadd_pd(
        _mm256_load_pd(a+i+k*N),
        _mm256_broadcast_sd(b+k+j*N),
        c0);
}
_mm256_store_pd(c+i+j*N, c0);
```
“Vectorized” dgemm

```c
// AVX intrinsics; P&H p. 227
void dgemm_avx(int N, double *a, double *b, double *c) {
   // avx operates on 4 doubles in parallel
   for (int i=0; i<N; i+=4) {
      for (int j=0; j<N; j++) {
         // c0 = c[i][j]
         __m256d c0 = {0,0,0,0};
         for (int k=0; k<N; k++) {
            c0 = _mm256_add_pd(
               c0, // c0 += a[i][k] * b[k][j]
               _mm256_mul_pd(
                  _mm256_load_pd(a+i+k*N),
                  _mm256_broadcast_sd(b+k+j*N)));
         }
         _mm256_store_pd(c+i+j*N, c0); // c[i,j] = c0
      }
   }
}
```
Performance

<table>
<thead>
<tr>
<th>N</th>
<th>Gflops</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>scalar</td>
</tr>
<tr>
<td>32</td>
<td>1.30</td>
</tr>
<tr>
<td>160</td>
<td>1.30</td>
</tr>
<tr>
<td>480</td>
<td>1.32</td>
</tr>
<tr>
<td>960</td>
<td>0.91</td>
</tr>
</tbody>
</table>

- 4x faster
- But still << theoretical 25 GFLOPS!
Agenda

- 61C – the big picture
- Parallel processing
- Single instruction, multiple data
- SIMD matrix multiplication
- **Loop unrolling**
- Memory access strategy - blocking
- And in Conclusion, ...
Loop Unrolling

- On high performance processors, optimizing compilers performs “loop unrolling” operation to expose more parallelism and improve performance:

```c
for(i=0; i<N; i++)
    x[i] = x[i] + s;
```

- Could become:

```c
for(i=0; i<N; 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;
}
```

1. Expose data-level parallelism for vector (SIMD) instructions or superscalar multiple instruction issue

2. Mix pipeline with unrelated operations to help with reduce hazards

3. Reduce loop “overhead”

4. Makes code size larger
Amdahl’s Law* applied to `dgemm`

- Measured `dgemm` performance
  - Peak: 5.5 GFLOPS
  - Large matrices: 3.6 GFLOPS
  - Processor: 24.8 GFLOPS

- Why are we not getting (close to) 25 GFLOPS?
  - Something else (not floating-point ALU) is limiting performance!
  - But what? Possible culprits:
    - Cache
    - Hazards
    - Let’s look at both!

---

* Amdahl’s Law states that the amount a speedup possible through parallelism is limited by the sequential (non-parallel) work.
Pipeline Hazards – \texttt{dgemm}

```
// AVX intrinsics; P&H p. 227
void \texttt{dgemm\_avx}(\texttt{int N, double \*a, double \*b, double \*c}) { 
    // avx operates on 4 doubles in parallel
    for (\texttt{int i=0; i<N; i+=4}) {
        for (\texttt{int j=0; j<N; j++}) {
            \texttt{c0 = c[i][j]}
            \texttt{m256d c0 = \{0,0,0,0\};}
            for (\texttt{int k=0; k<N; k++}) {
                \texttt{c0 = _mm256\_add\_pd(c0, \_mm256\_mul\_pd(c0, _mm256\_load\_pd(a+i+k*N),
                                                                 _mm256\_broadcast\_sd(b+k+j*N))));}
            }
            \texttt{_mm256\_store\_pd(c+i+j*N, c0); // c[i,j] = c0}
        }
    }
}
```

“add\_pd” depends on result of “mult\_pd” which depends on “load\_pd”
Loop Unrolling

How do you verify that the generated code is actually unrolled?
## Performance

<table>
<thead>
<tr>
<th>N</th>
<th>Gflops</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>scalar</td>
<td>avx</td>
<td>unroll</td>
</tr>
<tr>
<td>32</td>
<td>1.30</td>
<td>4.56</td>
<td>12.95</td>
</tr>
<tr>
<td>160</td>
<td>1.30</td>
<td>5.47</td>
<td>19.70</td>
</tr>
<tr>
<td>480</td>
<td>1.32</td>
<td>5.27</td>
<td>14.50</td>
</tr>
<tr>
<td>960</td>
<td>0.91</td>
<td>3.64</td>
<td>6.91</td>
</tr>
</tbody>
</table>

WOW!
Agenda

• 61C – the big picture
• Parallel processing
• Single instruction, multiple data
• SIMD matrix multiplication
• Amdahl’s law
• Loop unrolling
• Memory access strategy - blocking
• And in Conclusion, ...
FPU versus Memory Access

• How many floating-point operations does matrix multiply take?
  – $F = 2 \times N^3$ (N^3 multiplies, N^3 adds)

• How many memory load/stores?
  – $M = 3 \times N^2$ (for A, B, C)

• Many more floating-point operations than memory accesses
  – $q = \frac{F}{M} = \frac{2}{3} \times N$
  – Good, since arithmetic is faster than memory access
  – Let’s check the code ...
But memory is accessed repeatedly

**Inner loop:**

```c
for (int k=0; k<N; k++) {
    c0 = _mm256_add_pd(
        c0, // c0 += a[i][k] * b[k][j]
        _mm256_mul_pd(
            _mm256_load_pd(a+i+k*N),
            _mm256_broadcast_sd(b+k+j*N)));
}
```

- \( q = \frac{F}{M} = 1.6! \) (1.25 loads and 2 floating-point operations)
Typical Memory Hierarchy

- Where are the operands (A, B, C) stored?
- What happens as N increases?
- **Idea:** arrange that most accesses are to fast cache!
Blocking

• Idea:
  – Rearrange code to use values loaded in cache many times
  – Only “few” accesses to slow main memory (DRAM) per floating point operation
  – $\Rightarrow$ throughput limited by FP hardware and cache, not slow DRAM
  – P&H, RISC-V edition p. 465
Blocking Matrix Multiply
(divide and conquer: sub-matrix multiplication)
Memory Access Blocking

```c
// Cache blocking; P&H p. 555
const int BLOCKSIZE = 32;

void do_block(int n, int si, int sj, int sk, double *A, double *B, double *C) {
    for (int i=si; i<i+BLOCKSIZE; i+=UNROLL*4) {
        for (int j=sj; j<sj+BLOCKSIZE; j++) {
            _m256d c[4];
            for (int x=0; x<UNROLL; x++)
                c[x] = _mm256_load_pd(C+i+x+r+j+n);
            for (int k=sk; k<sk+BLOCKSIZE; k++) {
                _m256d b = _mm256_broadcast_sd(B+k+j*n);
                for (int x=0; x<UNROLL; x++)
                    c[x] = _mm256_add_pd(c[x],
                                        _mm256_mul_pd(_mm256_load_pd(A+n+k+x+r+i), b));
            }
            for (int x=0; x<UNROLL; x++)
                _mm256_store_pd(C+i+x+r+j*n, c[x]);
        }
    }
}

void dgemm_block(int n, double* A, double* B, double* C) {
    for (int sj=0; sj<n; sj+=BLOCKSIZE)
        do_block(n, si, sj, sk, A, B, C);
    for (int sk=0; sk<n; sk+=BLOCKSIZE)
        do_block(n, si, sj, sk, A, B, C);
}
```
## Performance

<table>
<thead>
<tr>
<th>N</th>
<th>scalar</th>
<th>avx</th>
<th>unroll</th>
<th>blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1.30</td>
<td>4.56</td>
<td>12.95</td>
<td>13.80</td>
</tr>
<tr>
<td>160</td>
<td>1.30</td>
<td>5.47</td>
<td>19.70</td>
<td>21.79</td>
</tr>
<tr>
<td>480</td>
<td>1.32</td>
<td>5.27</td>
<td>14.50</td>
<td>20.17</td>
</tr>
<tr>
<td>960</td>
<td>0.91</td>
<td>3.64</td>
<td>6.91</td>
<td>15.82</td>
</tr>
</tbody>
</table>
Agenda

• 61C – the big picture
• Parallel processing
• Single instruction, multiple data
• SIMD matrix multiplication
• Amdahl’s law
• Loop unrolling
• Memory access strategy - blocking
• And in Conclusion, ...
And in Conclusion, ...

• Approaches to Parallelism
  – SISD, SIMD, MIMD (next lecture)

• SIMD
  – One instruction operates on multiple operands simultaneously

• Example: matrix multiplication
  – Floating point heavy \( \rightarrow \) exploit Moore’s law to make fast