Parallelism 1
And Switching Gears: GPIO

• We see how to do high performance I/O
  • CPU has data it wants to send in main memory
  • Configures device & DMA controller to initiate transfer
    • Device then receives the data through DMA

• We have moderate bandwidth, flexible I/O
  • Universal Serial Bus is really a lightweight network for your slower peripheral devices

• But what about human scale?
  • With people, we only need to react in milliseconds to hours
Reminder: Amdahl's Law and Programming Effort

- Don't optimize where you don't need to
  - And if I only need to react at kHz granularity...
    But my processor is a GHz...
- I have 1 million clock cycles to actually decide what to do!
- So let's provide a simple interface
Raspberry Pi GPIO

• A set of physical pins hooked up to the CPU
• The CPU can write and read these pins as memory, like any other I/O device
• But that is a low level pain for us humans...
• So the Linux installation provides "files" that can access GPIO
• You can literally write a 1 or a 0 to a pin or read the value at a pin
• Plus faster & still simple APIs
Using GPIO

• There are a lot of add-on cards...
  • EG, ones for controlling servos

• Or you can build your own

• Combined with USB provides very powerful glue...

• Similarly some even smaller devices:
  • Adafruit "Trinket": 8 MHz 8-bit microcontroller, 5 GPIO pins
    Get it for $8 at the Jacobs Hall store...

• Big application: Serial LED strings
  • Color LEDs that have a bit-serial interface
And "Me Bad" on Low Rate I/O...

- I said for low rate (mice, etc) interrupts make the most sense
  - Polling overhead is low, but so is interrupt overhead...
- But that makes sense only if the devices support interrupts
  - But our keyboards and mice don't actually support interrupts!
- Old days: PS2 interface directly plugged in a keyboard or mouse
  - And the keyboard/mouse generated an interrupt on keypress or movement
  - No overhead if nothing going on...
But recently I learned that USB doesn't actually support interrupts!!!

- USB is a tree
  - With the controller at the root...
  - Devices are addressed by the path through the tree

- For simplicity, the controller controls **everything**
  - It sends out requests to the devices to read/write data
  - An "interrupt device" (low rate, high priority device like a keyboard or mouse) doesn't actually send interrupts!
    - So no "send and listen" like our networks

- Instead, every request starts with the controller
  - And it just guarantees that it will be able to poll the devices at a given rate...
    It schedules any bulk transfers so that it still has time to ask for data

- Thus USB uses polling, **because it doesn't have the ability to generate interrupts!**
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:
  • Binary numbers
  • C
  • Pointers
  • Assembly language
  • Processor micro-architecture
  • Pipelining
  • Caches
  • 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[][]` in C uses row-major
- Fortran uses column-major
- Our examples use column-major

\[
a_{ij} : a[i \times N + j] \quad a_{ij} : a[i + j \times N]
\]
Simplifying Assumptions…

• We want to keep the examples (somewhat) manageable…

• We will keep the matrixes square
  • So both matrixes are the same size
    with the same number of rows and columns

• We will keep the matrixes reasonably aligned
  • So size % a reasonable power of 2 == 0
**dgemm** Reference Code: 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 2\*N³ flops
c = a * b
a, b, c are N x N matrices

// 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;
        }
}
#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!
But This Is Part 1 of Project 4...

- You are developing a C-based library for python
  - dumbpy

- First is going to be implementing the C functions in sequential code
  - So right here the goal is the 200x speedup over naive python

- We've received complaints that 61C students aren't good enough C coders...
  - So we have structured project 4 to require a fair amount of C writing before you get to the parallel coding...
And Overall Administrivia...

• https://www.yellkey.com/itself
• Project 4 due out RSN...
• Performance homework out now
  • Probably one final homework after that
• Tentatively scheduling a project party for Thursday
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 …)
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
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
  - We consider superscalar as SISD because the programming model is sequential

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)
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 22 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.
Flynn* Taxonomy, 1966

- 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

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

*Prof. Michael Flynn, Stanford
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
  • Video cards
First SIMD Extensions: MIT Lincoln Labs TX-2, 1957
Intel x86 SIMD: Continuous Evolution

**MMX 1997**

- **1999**
  - SSE
  - 70 instr
  - Single-Precision Vectors
  - Streaming operations

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

- **2004**
  - SSE3
  - 13 instr
  - Complex Data

- **2006**
  - SSSE3
  - 32 instr
  - Decode

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

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

- **2009**
  - AES-NI
  - 7 instr
  - Encryption and Decryption
  - Key Generation

- **2010/11**
  - AVX
  - ~100 new instr.
  - ~300 legacy sse instr updated
  - 256-bit vector
  - 3 and 4-operands instructions
<table>
<thead>
<tr>
<th>Year</th>
<th>GFLOPS</th>
<th>Processor</th>
<th>Technology</th>
<th>SIMD Features</th>
<th>Memory</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>87</td>
<td>Westmere</td>
<td>32 nm</td>
<td>SSE 4.2</td>
<td>DDR3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AVX 256 bit</td>
<td>PCIe2</td>
<td></td>
</tr>
<tr>
<td>2012</td>
<td>185</td>
<td>Sandy Bridge</td>
<td>32 nm</td>
<td>AVX 256 bit</td>
<td>DDR3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(new registers)</td>
<td>PCIe3</td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>~225</td>
<td>Ivy Bridge</td>
<td>22 nm</td>
<td>AVX2 512 bit</td>
<td>DDR4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(new instructions)</td>
<td>PCIe3</td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>~500</td>
<td>Haswell</td>
<td>22 nm</td>
<td>AVX2 512 bit</td>
<td>DDR4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(new instructions)</td>
<td>PCIe3</td>
<td></td>
</tr>
<tr>
<td>2015</td>
<td>TBD</td>
<td>Broadwell</td>
<td>14 nm</td>
<td>AVX3 512 bit</td>
<td>DDR4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(new registers)</td>
<td>PCIe4</td>
<td></td>
</tr>
<tr>
<td>Future</td>
<td>TBD</td>
<td>Skylake</td>
<td>14 nm</td>
<td>AVX3 512 bit</td>
<td>DDR4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(new registers)</td>
<td>PCIe4</td>
<td></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:
machdep.cpu.brand_string: Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz

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: RDWRFSGS TSC_THREAD_OFFSET BMI1 AVX2 SMEX
BMI2 ERMS INVPICID FPU_CSDS RDSEED ADX_SMAP IPT MDCLEAR IBRS STIBP
L1DF SSBD
```
AVX SIMD Registers: Greater Bit Extensions Overlap Smaller Versions
Intel SIMD Data Types

(Now also AVX-512 available (but not on Hive): 16x float and 8x double)
Agenda

• 61C – the big picture
• Parallel processing
• Single instruction, multiple data
• **SIMD matrix multiplication**
• 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 …
  - Or to google, either one…
- Can we use the compiler to generate all non-SIMD instructions?
x86 SIMD “Intrinsics”

The Intel Intrinsics Guide is an interactive reference tool for Intel intrinsic instructions, which are C style functions that provide access to many Intel instructions - including Intel* SSE, AVX, AVX-512, and more - without the need to write assembly code.

4 parallel multiplies
2 instructions per clock cycle (CPI = 0.5)
4 cycles latency (data hazard time...)

mul_pd

__m256d _mm256_mul_pd (__m256d a, __m256d b)

Synopsis
__m256d _mm256_mul_pd (__m256d a, __m256d b)
#include <immintrin.h>
Instruction: vmulpd ymm, ymm, ymm
CPUID Flags: AVX

Description
Multiply packed double-precision (64-bit) floating-point elements in a and b, and store the results in dst.

Operation
FOR j := 0 to 3
  i := j*64
ENDFOR
dst[MAX:256] := 0

Performance
<table>
<thead>
<tr>
<th>Architecture</th>
<th>Latency</th>
<th>Throughput (CPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Icecake</td>
<td>4</td>
<td>0.5</td>
</tr>
<tr>
<td>Skylake</td>
<td>4</td>
<td>0.5</td>
</tr>
<tr>
<td>Broadwell</td>
<td>3</td>
<td>0.5</td>
</tr>
<tr>
<td>Haswell</td>
<td>5</td>
<td>0.5</td>
</tr>
</tbody>
</table>
x86 Intrinsics AVX Data Types

**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 <em>nnn</em>, where <em>nnn</em> 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/
Vectorized Matrix Multiplication

Inner Loop:

```c
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, _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>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>scalar</td>
<td>avx</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1.30</td>
<td>4.56</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>1.30</td>
<td>5.47</td>
<td></td>
</tr>
<tr>
<td>480</td>
<td>1.32</td>
<td>5.27</td>
<td></td>
</tr>
<tr>
<td>960</td>
<td>0.91</td>
<td>3.64</td>
<td></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 perform “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 super-scalar 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 \texttt{dgemm}

- Measured \texttt{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!
“Vectorized” dgemm: Pipeline Hazards

```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
        }
    }
}
```

“add_pd” depends on result of “mult_pd” which depends on “load_pd”
Loop Unrolling

// Loop unrolling; P&H p. 352
const int UNROLL = 4;

void dgemm_unroll(int n, double *A, double *B, double *C) {
    for (int i=0; i<n; i+= UNROLL*4) {
        for (int j=0; j<n; j++) {
            _m256d c[4];
            for (int x=0; x<UNROLL; x++)
                c[x] = _mm256_load_pd(C+i+x*4+j*n);
            for (int k=0; k<n; 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*4+i), b));
            }
        }
    }
}

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

<table>
<thead>
<tr>
<th>N</th>
<th>scalar</th>
<th>avx</th>
<th>unroll</th>
</tr>
</thead>
<tbody>
<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
  - \( q = F/M = 2/3 \times N \)
  - Good, since arithmetic is faster than memory access
  - Let’s check the code …
But memory is accessed repeatedly

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

**Inner loop:**

```c
for (int k=0; k<N; k++) {
    c0 = _mm256_add_pd(c0, \( a[i][k] \times b[k][j] \);
    _mm256_load_pd(a+i+k*N),
    _mm256_broadcast_sd(b+k+j*N));
}
```
- 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
    - -> 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. 556
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<si+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*4+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*4+i), b));
            }
            for (int x=0; x<UNROLL; x++)
                __mm256_store_pd(C+i+x*4+j*n, c[x]);
        }
    }
}

void dgemm_block(int n, double* A, double* B, double* C) {
    for (int sj=0; sj<n; sj+=BLOCKSIZE)
        for (int si=0; si<n; si+=BLOCKSIZE)
            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>
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 -> exploit Moore’s law to make fast