Networking & Parallelism 1
Networks: Talking to the Outside World

• Originally sharing I/O devices between computers
  • E.g., printers

• Then communicating between computers
  • E.g., file transfer protocol

• Then communicating between people
  • E.g., e-mail

• Then communicating between networks of computers
  • E.g., file sharing, www, …

• Then turning multiple cheap systems into a single computer
  • Warehouse scale computing
The Internet (1962)

• History
  • 1963: JCR Licklider, while at DoD’s ARPA, writes a memo describing desire to connect the computers at various research universities: Stanford, Berkeley, UCLA, ...
  • 1969: ARPA deploys 4 “nodes” @ UCLA, SRI, Utah, & UCSB
  • 1973 Robert Kahn & Vint Cerf invent TCP, now part of the Internet Protocol Suite

• Internet growth rates
  • Exponential since start
  • But finally starting to hit human scale limits although lots of silicon cockroaches...

www.computerhistory.org/internet_history
www.greatachievements.org/?id=3736
en.wikipedia.org/wiki/Internet_Protocol_Suite

en.wikipedia.org/wiki/History_of_the_World_Wide_Web

- “System of interlinked hypertext documents on the Internet”
- History
  - 1945: Vannevar Bush describes hypertext system called “memex” in article
  - 1989: Sir Tim Berners-Lee proposed and implemented the first successful communication between a Hypertext Transfer Protocol (HTTP) client and server using the internet.
  - 1993: NCSA Mosaic: A graphical HTTP client
  - ~2000 Dot-com entrepreneurs rushed in, 2001 bubble burst
- Today: Access anywhere!
Shared vs. Switch-Based Networks

- **Shared vs. Switched:**
  - **Shared:** 1 at a time (CSMA/CD)
  - **Switched:** pairs ("point-to-point" connections) communicate at same time

- Aggregate bandwidth (BW) in switched network is many times that of shared:
  - point-to-point faster since no arbitration, simpler interface
  - Wired is almost always switched
  - Wireless is by definition shared
Shared Broadcast

• Old-school Ethernet and Wireless
  • It doesn't just share but all others can see the request?

• How to handle access:
  • Old when I was old skool: Token ring
    • A single "token" that is passed around
  • Ethernet:
    • Listen and send
    • Randomized retry when someone else interrupts you
  • Cable Modem:
    • "Request to send": small request with a listen and send model
    • Big transfers then arbitrated
What makes networks work?

- links connecting switches and/or routers to each other and to computers or devices

- ability to name the components and to route packets of information - messages - from a source to a destination

- Layering, redundancy, protocols, and encapsulation as means of abstraction (61C big idea)
Software Protocol to Send and Receive

• **SW Send steps**
  - 1: Application copies data to OS buffer
  - 2: OS calculates checksum, starts timer
  - 3: OS sends DMA request to network interface HW and says start

• **SW Receive steps**
  - 3: Network interface copies data from network interface HW to OS buffer, triggers interrupt
  - 2: OS calculates checksum, if OK, send ACK; if not, delete message (sender resends when timer expires)
  - 1: If OK, OS copies data to user address space, & signals application to continue

<table>
<thead>
<tr>
<th>Dest</th>
<th>Src</th>
<th>Checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Net ID</td>
<td>Net ID</td>
<td>Len</td>
</tr>
<tr>
<td>CMD/ Address /Data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Header**
- **Payload**
- **Trailer**
Protocols for Networks of Networks?

What does it take to send packets across the globe?

- Bits on wire or air
- Packets on wire or air
- Delivery packets within a single physical network
- Deliver packets across multiple networks
- Ensure the destination received the data
- Create data at the sender and make use of the data at the receiver
The OSI 7 Layer Network Model

- A conceptual "Stack"
  - Physical Link: eg, the wires/wireless
  - Data Link: Ethernet
  - Network Layer: IP
  - Transport Layer: TCP/UDP
  - Session Layer/Presentation Layer/Application Layer
    - Session Layer/Presentation Layer really never got used
    - Political Layer: "Feinstein/Burr 'thou shalt not encrypt"
    - Nick is starting to spend way too much time on "layer 8" issues
Protocol Family Concept

- Protocol: packet structure and control commands to manage communication
- Protocol families (suites): a set of cooperating protocols that implement the network stack
- Key to protocol families is that communication occurs logically at the same level of the protocol, called peer-to-peer… …but is implemented via services at the next lower level
- Encapsulation: carry higher level information within lower level “envelope”
Inspiration…

• CEO Alice writes letter to CEO Bob
  • Folds letter and hands it to assistant

• Assistant:
  • Puts letter in envelope with CEO Bob’s full name
  • Takes to FedEx

• FedEx Office
  • Puts letter in larger envelope
  • Puts name and street address on FedEx envelope
  • Puts package on FedEx delivery truck

• FedEx delivers to other company

Dear Bob,

Your days are numbered.

--Alice
"Peers" on each side understand the same things
No one else needs to
Lowest level has most packaging
The Path Through FedEx

Deepest Packaging (Envelope+FE+Crate) at the Lowest Level of Transport
Protocol Family Concept

Each lower level of stack “encapsulates” information from layer above by adding header and trailer.
Most Popular Protocol for Network of Networks

- Transmission Control Protocol/Internet Protocol (TCP/IP)
- This protocol family is the basis of the Internet, a WAN (wide area network) protocol
  - IP makes best effort to deliver
    - Packets can be lost, corrupted
      - But corrupted packets should be turned into lost packets
  - TCP guarantees **reliable, in-order** delivery of a **bytestream**
    - Programs don't see packets, they just read and write strings of bytes
  - TCP/IP so popular it is used even when communicating locally: even across homogeneous LAN (local area network)
TCP/IP packet, Ethernet packet, protocols

- Application sends message
  - TCP breaks into 64KiB segments*, adds 20B header
  - IP adds 20B header, sends to network
  - If Ethernet, broken into 1500B packets with headers, trailers

* Not really. Because of fragments not always working, most TCP packets are sized so things fit in an Ethernet packet. That whole layering business is not as clean as we like...
TCP and UDP
The Two Internet Transfer Protocols

• **TCP: Transmission Control Protocol**
  • Connection based
    • SYN->SYN/ACK->ACK 3-way handshake
  • In order & reliable
    • All data is acknowledged
    • Programming interface is streams of data

• **UDP: Universal Datagram Protocol**
  • Datagram based
    • Just send messages
  • Out of order & unreliable
    • Datagrams can arrive in any order or be dropped (but not corrupted)
  • Needed for realtime protocols
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 not-quite-a-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 lets provide a simple interface
  • Because lets face it, my time is more valuable than the computer's time!
  • After all, 1 second of my time is worth 1,000,000,000 instructions!
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
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 $N \times N$
  - $i$ 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

$aij$:

Row-Major

<table>
<thead>
<tr>
<th>a00</th>
<th>a01</th>
<th>a02</th>
<th>a03</th>
</tr>
</thead>
<tbody>
<tr>
<td>a10</td>
<td>a11</td>
<td>a12</td>
<td>a13</td>
</tr>
<tr>
<td>a20</td>
<td>a21</td>
<td>a22</td>
<td>a23</td>
</tr>
<tr>
<td>a30</td>
<td>a31</td>
<td>a32</td>
<td>a33</td>
</tr>
</tbody>
</table>

Column-Major

<table>
<thead>
<tr>
<th>a13</th>
</tr>
</thead>
<tbody>
<tr>
<td>a12</td>
</tr>
<tr>
<td>a11</td>
</tr>
<tr>
<td>a10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a03</th>
</tr>
</thead>
<tbody>
<tr>
<td>a02</td>
</tr>
<tr>
<td>a01</td>
</tr>
<tr>
<td>a00</td>
</tr>
</tbody>
</table>

Row

$aij : a[i*N + j]$  
$aij : a[i + j*N]$
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
```
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
// 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++)
                // \( a[i][k] \times b[k][j] \)
                cij += a[i+k*N] * b[k+j*N];
            // \( c[i][j] \)
            c[i+j*N] = cij;
        }
}
```
Timing Program Execution

```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;
}
```
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!
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

<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>

- 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*
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 Mode:

Scalar Mode:

A + B

A

B

A + B
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-operand instructions
### Intel Advanced Vector eXtensions

<table>
<thead>
<tr>
<th>Year</th>
<th>2011</th>
<th>2012</th>
<th>2013</th>
<th>2014</th>
<th>2015</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Westmere</td>
<td>Sandy Bridge</td>
<td>Ivy Bridge</td>
<td>Haswell</td>
<td>Broadwell</td>
<td>Skylake</td>
</tr>
<tr>
<td>GFLOPS</td>
<td>87</td>
<td>185</td>
<td>~225</td>
<td>~500</td>
<td>tbd</td>
<td>tbd</td>
</tr>
<tr>
<td>Process</td>
<td>32 nm</td>
<td>32 nm</td>
<td>22 nm</td>
<td>22 nm</td>
<td>14 nm</td>
<td>14 nm</td>
</tr>
<tr>
<td>Features</td>
<td>SSE 4.2, DDR3, PCIe2</td>
<td>AVX (256 bit registers), DDR3, PCIe3</td>
<td></td>
<td>AVX2 (new instructions), DDR4, PCIe3</td>
<td></td>
<td>AVX 3.2 (512 bit registers), DDR4, 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: 4
hw.logicalcpu: 8

machdep.cpu.brand_string: Intel(R) Core(TM) i5-1038NG7 CPU @ 2.00GHz

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 SGX BMI1 AVX2 FDPEO SMEP BMI2 ERMS INVPACID FPU_CSDS AVX512F AVX512DQ RDSEED ADX SMAP AVX512IFMA CLFSOPT IPT AVX512CD SHA AVX512BW AVX512VL AVX512VBMI UMIP PKU GFNI VAES VPCLMULQDQ AVX512VNNI AVX512BITALG AVX512VPOPCNDQ RDPID SGXLC FSREPMOV MDCLEAR IBRS STIBP L1DF ACAPMSR SSBD

machdep.cpu.extfeatures: SYSCALL XD 1GBPAGE EM64T LAHF LZCNT PREFETCHW RDTSCP TSCI
AVX SIMD Registers: Greater Bit Extensions Overlap Smaller Versions
(AVX-512 available (but not on Hive so you can't use on Proj 4):
16x float and 8x double)...
But latest: Intel has decided to basically give up on AVX-512 going forward!
Alder Lake's "efficient" cores don't include it so it is turned off!
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"
**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/
Vectorized Matrix Multiplication

**Inner Loop:**

```c
for i ...; i+=4
```

```
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>
<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 \hspace{1cm} 5.5 GFLOPS
  - Large matrices \hspace{1cm} 3.6 GFLOPS
  - Processor \hspace{1cm} 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

```c
// 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
  - \( q = \frac{F}{M} = \frac{2}{3} \times N \)
  - Good, since arithmetic is faster than memory access
  - Let’s check the code …
But memory is accessed repeatedly

- $q = \frac{F}{M} = 1.6!$ (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] * b[k][j] * m256_load_sd(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