Reference Problem

• Matrix multiplication
  – Basic operation in many engineering, data, and imaging processing tasks
  – Image filtering, noise reduction, ...
  – Many closely related operations
    ▪ E.g. stereo vision (project 4)

• dgemm
  – Double precision floating point matrix multiplication
Application Example: Deep Learning

• Image classification (cats ...)
• Pick “best” vacation photos
• Machine translation
• Clean up accent
• Fingerprint verification
• Automatic game playing
Matrices

- Square (or rectangular) $N \times N$ array of numbers
  - Dimension $N$

\[
C = A \cdot B
\]

\[
c_{ij} = \sum_{k} a_{ik} b_{kj}
\]
Matrix Multiplication

\[ C = A \cdot B \]

\[ c_{ij} = \sum_{k} a_{ik} b_{kj} \]
Reference: Python

- 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 $2*N^3$ 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++)
                cij += a[i+k*N] * b[k+j*N];
            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;
}

Timing Program Execution
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 class gives you this kind of power?
We could stop here ... but why? Let’s do better!
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**

**Hardware**

- Warehouse Scale Computer

- Harness Parallelism & Achieve High Performance

**Today’s Lecture**

- Computer
  - Core
  - Memory
  - Input/Output
  - Instruction Unit(s)
  - Functional Unit(s)

- Cache Memory

- Logic Gates

- Smart Phone

- Warehouse

- Scale

- Computer

- Core

- Memory

- Input/Output

- Functional Unit(s)

- Cache Memory

- Logic Gates

- Smart Phone
Multiple-Instruction/Single-Data Stream (MISD)

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

This has few applications. Not covered in 61C.
SIMD Applications & Implementations

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

• Implementations
  – x86
  – ARM
  – …
Raw Double Precision Throughput (Bernhard’s Powerbook Pro)

<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

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>
<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 $\ll$ theoretical 25 Gflops!
Pipeline Hazards – \texttt{dgemm}

\textbf{ Technologies }
- [ ] MMX
- [ ] SSE
- [ ] SSE2
- [ ] SSE3
- [ ] SSSE3
- [ ] SSE4.1
- [ ] SSE4.2
- [x] AVX
- [ ] AVX2
- [ ] FMA
- [ ] AVX-512
- [ ] KNC
- [ ] SVML
- [ ] Other

\textbf{ Categories }
- [ ] Application-Targeted
- [ ] Arithmetic
- [ ] Bit Manipulation
- [ ] Cast
- [ ] Compare

\begin{verbatim}
\texttt{mul_pd}

\_\texttt{m256d} \_\texttt{mm256\_mul\_pd} (\_\texttt{m256d} a, \_\texttt{m256d} b)

\textbf{ Synopsis }
\_\texttt{m256d} \_\texttt{mm256\_mul\_pd} (\_\texttt{m256d} a, \_\texttt{m256d} b)
#include "immintrin.h"
Instruction: vmulpd ymm, ymm, ymm
CPUID Flags: AVX

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

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

\textbf{ Performance }
\begin{tabular}{|c|c|c|}
\hline
Architecture & Latency & Throughput \\
\hline
Haswell & 5 & 0.5 \\
Ivy Bridge & 5 & 1 \\
Sandy Bridge & 5 & 1 \\
\hline
\end{tabular}
\end{verbatim}
// 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));
            }
            for (int x=0; x<UNROLL; x++)
                _mm256_store_pd(C+i+x*4+j*n, c[x]);
        }
    }  
}

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>