Parallelism 1
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

- Dense 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
    - 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[i][j]` in C uses row-major
- Fortran uses column-major
- Our examples use column-major

<table>
<thead>
<tr>
<th>i</th>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a00</td>
<td>a01</td>
<td>a02</td>
<td>a03</td>
</tr>
<tr>
<td>1</td>
<td>a10</td>
<td>a11</td>
<td>a12</td>
<td>a13</td>
</tr>
<tr>
<td>2</td>
<td>a20</td>
<td>a21</td>
<td>a22</td>
<td>a23</td>
</tr>
<tr>
<td>3</td>
<td>a30</td>
<td>a31</td>
<td>a32</td>
<td>a33</td>
</tr>
</tbody>
</table>

\[ a_{ij} \]: `a[i*N + j]`

Row-Major

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a00</td>
<td>a10</td>
<td>a20</td>
<td>a30</td>
</tr>
<tr>
<td>1</td>
<td>a01</td>
<td>a11</td>
<td>a21</td>
<td>a31</td>
</tr>
<tr>
<td>2</td>
<td>a02</td>
<td>a12</td>
<td>a22</td>
<td>a32</td>
</tr>
<tr>
<td>3</td>
<td>a03</td>
<td>a13</td>
<td>a23</td>
<td>a33</td>
</tr>
</tbody>
</table>

Column-Major

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a00</td>
<td>a10</td>
<td>a20</td>
<td>a30</td>
</tr>
<tr>
<td>1</td>
<td>a01</td>
<td>a11</td>
<td>a21</td>
<td>a31</td>
</tr>
<tr>
<td>2</td>
<td>a02</td>
<td>a12</td>
<td>a22</td>
<td>a32</td>
</tr>
<tr>
<td>3</td>
<td>a03</td>
<td>a13</td>
<td>a23</td>
<td>a33</td>
</tr>
</tbody>
</table>

\[ a_{ij} \]: `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
- We are doing dense matrix multiplication
  - A related problem is "sparse" matrix multiplication, where most of the entries are 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 $2*N^3$ flops
c = a * b

a, b, c are $N \times N$ matrices

```c
void dgemm_scalar(int N, double *a, double *b, double *c){
    int i,j,k; double cij;
    for(i = 0; i < N; ++i){
        for(j = 0; j < N; ++j){
            cij = 0
            for(k = 0; k < N; ++k){
                cij += a[i+k*N] * b[k+j*N];
            }
            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) / CLOCKKS_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 \textit{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
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

\[
\begin{align*}
\text{SIMD Mode} & \quad + \\
A_7 & \quad A_6 & \quad A_5 & \quad A_4 & \quad A_3 & \quad A_2 & \quad A_1 & \quad A_0 \\
B_7 & \quad B_6 & \quad B_5 & \quad B_4 & \quad B_3 & \quad B_2 & \quad B_1 & \quad B_0 \\
\text{= } & \quad \text{= } \\
A_7+B_7 & \quad A_6+B_6 & \quad A_5+B_5 & \quad A_4+B_4 & \quad A_3+B_3 & \quad A_2+B_2 & \quad A_1+B_1 & \quad A_0+B_0 \\
\text{Scalar Mode} & \quad + \\
A & \quad A \\
\text{= } & \quad \text{= } \\
A+B & \quad A+B
\end{align*}
\]
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

2011

Westmere
32 nm
SSE 4.2
DDR3
PCIe2

2012

Sandy Bridge
32 nm
AVX (256 bit registers)
DDR3
PCIe3

2013

Ivy Bridge
22 nm

2014

Haswell
22 nm
AVX2 (new instructions)
DDR4
PCIe3

2015

Broadwell
14 nm

Future

Skylake
14 nm
AVX 3.2 (512 bit registers)
DDR4
PCIe4

AVX Registers getting wider, instruction set getting richer

AVX also supported by AMD processors

Note on AVX-512...

- 512 bit vectors may be more efficient
  - Twice as much operations per instruction
  - But also requires 2x the resources over 256b

- Really heavy vector code is now done on the GPU
  - Thousands of simultaneous operations

- Intel is now disabling it on some of the newer designs
  - The big/little "Alder Lake":
    The little cores don't support AVX-512
    The big ones do... but have it disabled

- For us, the Hive is only AVX-2 anyway
  - So no big loss
$ 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
DSCLP 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 INVPICD FPU_CSDS AVX512F AVX512DQ RDSEED ADX SMAP AVX512IFMA CLFSEOPT IPT
AVX512CD SHA AVX512BW AVX512VL AVX512VBMI UMIP PKU GFNI VAES VPCLMULQDQ AVX512VNNI
AVX512BITALG AVX512VPQPCNDQ RDPID SGXLC FSREPMOV MDCLEAR IBRS STIBP L1DF ACAPMSR
SSBD

machdep.cpu.extfeatures: SYSCALL XD 1GBPAGE EM64T LAHF LZCNT PREFETCHW RDTSCP TSCI
AVX SIMD Registers: 16 registers, Greater Bit Extensions Overlap Smaller Versions
Intel SIMD Data Types

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!
Doggo Break!
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.

## mul_pd

### Synopsis

```c
__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

```c
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>Icelake</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>

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

for i ...; i+=4

Inner Loop: for j ...

```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
void dgemm_avx(int N, double *a, double *b, double *c){
    int i,j,k; __m256d c0;
    for(i = 0; i < N; i += 4){
        for(j = 0; j < N; ++j){
            c0 = __mm256_add_pd(c0,
                __mm256_mull_pd(
                    __mm256_load_pd(a+i+k*N),
                    __mm256_broadcast_sd(b+k+j*N)));
        }
        __mm256_store_pd(c+i+j*N, c0);
    }
}
```
### Performance

<table>
<thead>
<tr>
<th>N</th>
<th>Gflops scalar</th>
<th>Gflops avx</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1.30</td>
<td>4.56</td>
</tr>
<tr>
<td>160</td>
<td>1.30</td>
<td>5.47</td>
</tr>
<tr>
<td>480</td>
<td>1.32</td>
<td>5.27</td>
</tr>
<tr>
<td>960</td>
<td>0.91</td>
<td>3.64</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 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
void dgemm_avx(int N, double *a, double *b, double *c){
    int i,j,k; __m256d c0;
    for(i = 0; i < N; i += 4){
        for(j = 0; j < N; ++j){
            c0 = {0,0,0,0}
            for(k = 0; k < N; ++k){
                c0 = __mm256_add_pd(c0,
                                   __mm256_mull_pd(__mm256_load_pd(a+i+k*N),
                                                   __mm256_broadcast_sd(b+k+j*N)));
            }
            __mm256_store_pd(c+i+j*N, c0);
        }
    }
}
```

“add_pd” depends on result of “mult_pd” which depends on “load_pd”
Compiler does the unrolling

How do you verify that the generated code is actually unrolled?
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>
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) in the straightforward case

- 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

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

**Inner loop:**

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