CS250B: Modern Computer Systems
Performance Profiling with PerfTools

Sang-Woo Jun
How To Evaluate Our Approaches?

- Say, we made a performance engineering change in our program
  - ...And performance decreased by 10%
  - Why? Can we know?

- Many tools provide profiling capabilities
  - gprof, OProfile, Valgrind, VTune, PIN, ...

- We will talk about perf, part of perf tools
  - Native support in the Linux kernel
  - Straightforward PMC (Performance Monitoring Counter) support
Aside:
**Performance Monitoring Counters (PMC)**

- **Problem:** How can we measure architectural events?
  - L1 cache miss rates, branch mis-predicts, total cycle count, instruction count, ...
  - No way for software to know
  - Events happen too often for software to be counting them

- **Solution:** PMCs (Sometimes called Hardware Performance Counters)
  - Dozens of special registers that can each be programmed to count an event
  - Privileged registers, only accessible by kernel
  - Supported PMCs differ across models and designs

- **Usage**
  - Program PMC, read PMC, run piece of code, read PMC, compare read values
Linux Perf

- Performance analysis tool in Linux
  - Natively supported by kernel
  - Supports profiling a VERY wide range of events: PMC to kernel events
  - Note: needs sudo to do most things

- Many operation modes: top, stat, record, report, ...
  - Supported events found in “sudo perf list”
Linux Perf: Stat

- Default command prints some useful information
  - “sudo perf stat ls”

- More events can be traced using -e
  - sudo perf stat -e task-clock,page-faults,cycles,instructions,branches,branch-misses,LLC-loads,LLC-load-misses ls

```
Performance counter stats for 'ls':
  0.062008 task-clock (msec)  # 0.805 CPUs utilized
  0 context-switches         # 0.000 K/sec
  0 cpu-migrations          # 0.000 K/sec
  104 page-faults           # 1.69 M/sec
  2,797,861 cycles          # 4.29 GHz
  2,245,082 instructions     # 0.80 insn per cycle
  444,085 branches          # 681.119 M/sec
  16,749 branch-misses      # 3.77% of all branches
0.000810482 seconds time elapsed
```

```
Performance counter stats for 'ls':
  0.681485 task-clock (msec)  # 0.781 CPUs utilized
  102 page-faults            # 0.150 M/sec
  2,921,152 cycles           # 4.286 GHz
  2,217,325 instructions     # 0.76 insn per cycle
  439,589 branches          # 645.646 M/sec
  16,608 branch-misses      # 3.78% of all branches
  9,736 LLC-loads           # 14.286 M/sec
  3,269 LLC-load-misses     # 33.58% of all LLC cache hits
0.000872088 seconds time elapsed
```
Linux Perf: Record, Report

- Log events with “record”, interactively analyze it with “report”
  - `sudo perf record -e cycles,instructions,L1-dcache-loads,L1-dcache-load-misses [...]`
  - Creates “perf.data”

- “sudo perf report” reads “perf.data”

This is where most cycles are spent!

This is where most L1 cache misses are!
CS 250B: Modern Computer Systems
Modern Processors – Handling Branches

Sang-Woo Jun
What Do Conditionals Compile To?

- Conditionals (sometimes) compile to branch instructions in assembly
  - Compiler optimizations may replace branch instructions with something else
  - But not always

- Branch instructions take cycles(s)
  - At least one cycle, perhaps more
  - Obvious!

```java
int bar(int v) {
    if ( v == 0 ) return 1;
    else return 0;
}
```

```
bar(int):
push rbp
mov rbp, rsp
mov DWORD PTR [rbp-4], edi
cmp DWORD PTR [rbp-4], 0
jne .L2
mov eax, 1
jmp .L3
.L2:
mov eax, 0
.L3:
pop rbp
ret
```

gcc, x86-64, no optimizations

Generated using GCC explorer: https://gcc.godbolt.org/
Remember: Pipelined Processors and Hazards

- Modern, pipelined processors handle multiple instructions at once
  - Ideally, N-stage pipeline processes N instructions at a given cycle
  - But, sometimes future instructions depend on results of earlier ones ("Hazard")
  - Many types of hazards were introduced in undergrad architecture class

- Today, we look at the impact of handling "Control hazards"
Handling Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - e.g., Still working on decode stage of branch

```
i1: beq s0, zero, elsewhere
i2: addi s1, s0, 1
elsewhere:
i3: addi s1, s0, 2
```

Stalling until we know the correct answer results in multi-cycle overhead.
Control Hazard (Partial) Solution: Branch Prediction

- Processor will try to predict whether branch is taken or not
  - If prediction is correct, great!
    - Single cycle overhead
  - If not, we do not apply the effects of mis-predicted instructions
    - Effectively same performance penalty as stalling in this case
    - Can be many cycles of overhead depending on pipeline depth
Simple Branch Predictor Example

```
addi t1, zero, 3
addi t2, zero, 3
beq t1, t2, skip
sw t3, 0(t0)
ret

skip:
sw t2, 0(t0)
ret
```

Fetch -> Decode -> Execute -> Memory -> Writeback

- **Addi**
- **Addi**
- **Addi**
- **Addi**
- **Beq**
- **Addi**
- **Addi**
- **Addi**
- **Addi**
- **Addi**

Mispredict detected!

Pipeline bubbles

Fetch correct branch

No state update before Execute stage can detect misprediction (Fetch and Decode stages don’t write to register)
Some Classes Of Branch Predictors

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken

- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history (1-bit “taken” or “not taken”) of each branch in a fixed size “branch history table”
  - **Assume future behavior will continue the trend**
    - When wrong, stall while re-fetching, and update history

Many many different methods, Lots of research, some even using neural networks!
Branch prediction and performance

- Effectiveness of branch predictors is crucial for performance
  - Spoilers: On SPEC benchmarks, modern predictors routinely have 98+% accuracy
  - Of course, less-optimized code may have much worse behavior

- Branch-heavy software performance depends on good match between software pattern and branch prediction
  - Some high-performance software optimized for branch predictors in target hardware
  - Or, avoid branches altogether! (Branchless code)
Recap: Loop unrolling
A Compiler Solution To Branch Hazards

```
for ( i = 0 to 15 ) foo();
```

```
for ( i = 0 to 3 ) {
    foo();
    foo();
    foo();
    foo();
}
```

Loop unrolling

Potentially 16 branch mispredicts
Even without mispredicts,
branch instruction consume 16 cycles

Potentially 4 branch mis-predicts
Without mis-predicts,
branch instruction consume 4 cycles

We can do this manually, or tell the compiler to do its best
- GCC flags -funroll-loops, -funroll-all-loops
- How much to unroll depends on heuristics within compiler
Code Example: Counting Numbers

- How fast is the following code?
  - a and b are initialized to `rand()%256`
  - cnt is 100,000,000
  - Compiled with GCC –O3

```c
for ( int i = 0; i < cnt; i++ ) {
    if ( a[i] < 128 && b[i] < 128 ) lcnt++;
}
```

- This code takes 0.44s on my desktop (i5 @ 3 GHz)
  - Each loop takes 13.2 cycles (3 GHz * 0.44 / 100,000,000)
  - Can we do better? My x86 is 4-way superscalar!
Optimization Attempt #1: Loop Unrolling

- There are three potential branch instruction locations
  - “i < cnt”, “a[i] < 128”, and b[i] < 128”
- Is the bottleneck the “for” loop?
  - Let’s try giving -funroll-all-loops

```
for ( int i = 0; i < cnt; i++) {
    if ( a[i] < 128 && b[i] < 128 ) lcnt++;
}
```

- Performance increased from 0.44s to ~0.43s.
  - Better, but not by much
Identifying The Bottleneck

- We predict the “if” statements are the bottlenecks
  - Each of the two branch instructions has a 50% chance of being taken
  - Branch prediction very inefficient!

```c
for ( int i = 0; i < cnt; i++ ) {
  if ( a[i] < 128 && b[i] < 128 ) lcnt++;
}
```

- Performance improves when comparison becomes skewed
  - 0.44s when comparing against 128 (50%)
  - 0.27s when comparing against 64 (25%), 0.17s with 32
Optimization Attempt #2: Branchless Code

- Let’s try getting rid of the “if” statement. How?
- Some knowledge of architectural treatment of numbers is required
  - x86 represents negative numbers via two’s complement
  - “1” == 0x1, “-1” == 0xffffffff
  - “1>>31” == 0x0, “-1>>31” == 0xffffffff
- “(v-128)>>31”
  - if v >= 128: 0x0
  - v < 128: 0xffffffff

So many more instructions! Will this be faster?

```c
for ( int i = 0; i < cnt; i++ ) {
    lcnt += ( (((a[i] - 128)>>31)&1) * (((b[i] - 128)>>31)&1) );
}
```
## Comparing Performance Numbers

<table>
<thead>
<tr>
<th>Name</th>
<th>Elapsed (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vanilla</td>
<td>0.44 s</td>
</tr>
<tr>
<td>Branchless</td>
<td>0.06 s</td>
</tr>
</tbody>
</table>

~2 cycles per loop! 8 Operations with 4 way superscalar...

Branch predictor is almost always correct

Interestingly, loop with only one comparator is automatically optimized by compiler

```c
for (int i = 0; i < cnt; i++) {
    if (a[i] < 128) lcnt ++;
}
```

Shows same performance as the branchless one

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Command</th>
<th>Shared Object</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>87.38%</td>
<td>a.out</td>
<td>a.out</td>
<td>main</td>
</tr>
<tr>
<td>9.80%</td>
<td>a.out</td>
<td>libc-2.27.so</td>
<td>__random</td>
</tr>
<tr>
<td>1.48%</td>
<td>a.out</td>
<td>libc-2.27.so</td>
<td>__random_r</td>
</tr>
<tr>
<td>0.36%</td>
<td>a.out</td>
<td>[kernel.kallsyms] [k] __pagevec_lru_add_fn</td>
<td></td>
</tr>
<tr>
<td>0.29%</td>
<td>a.out</td>
<td>[kernel.kallsyms] [k] get_page_from_freelist</td>
<td></td>
</tr>
</tbody>
</table>

Vanilla: Total misses: 57 M out of 3,623 M

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Command</th>
<th>Shared Object</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>77.47%</td>
<td>a.out</td>
<td>libc-2.27.so</td>
<td>__random</td>
</tr>
<tr>
<td>10.13%</td>
<td>a.out</td>
<td>libc-2.27.so</td>
<td>__random_r</td>
</tr>
<tr>
<td>3.12%</td>
<td>a.out</td>
<td>[kernel.kallsyms] [k] get_page_from_freelist</td>
<td></td>
</tr>
<tr>
<td>2.86%</td>
<td>a.out</td>
<td>[kernel.kallsyms] [k] __pagevec_lru_add_fn</td>
<td></td>
</tr>
<tr>
<td>1.78%</td>
<td>a.out</td>
<td>[kernel.kallsyms] [k] __handle_mm_fault</td>
<td></td>
</tr>
<tr>
<td>0.74%</td>
<td>a.out</td>
<td>a.out</td>
<td>main</td>
</tr>
<tr>
<td>0.70%</td>
<td>a.out</td>
<td>libc-2.27.so</td>
<td>rand</td>
</tr>
</tbody>
</table>

Branchless: Total misses: 7 M out of 3,514 M

Over 7x performance!
Aside: Spectre (Simplified)

- Branch prediction is supposed to be transparent
- But not all aspects of it are!

```c
int array[256];
char forbidden = (*forbidden_ptr);
if ( a != a ) {
    int ignore = array[forbidden];
    Never executes, so never caught by sandbox (e.g., JVM, javascript)
    But mispredicted branch still affects cache!
}
for ( int i = 0; i < 256; i++ ) is_in_cache(array[i]);
Attacker can time the cache to discover contents
```
Slightly Deeper into Speculative Execution

- Branch prediction is one example of Speculative Execution
- Modern processors speculatively execute many things!
  - e.g., Does this Virtual Memory page belong to the Kernel?
  - Assume we have access to everything, and roll back state if turns out to be false
  - Always waiting for checks is too much overhead!
Aside: Meltdown (Simplified)

- Caches are supposed to be transparent
- But not all aspects of it are!

```c
int array[256];
char forbidden = (*forbidden_ptr);
int ignore = array[forbidden];
for ( int i = 0; i < 256; i++ ) is_in_cache(array[i]);
```

Assuming (*forbidden_ptr) returns zeros in the correct operation, instead of protection fault

Always being safe is too much overhead. How do we fix this?
Questions?
CS 250B: Modern Computer Systems

Modern Processors – SIMD Extensions

Sang-Woo Jun
Modern Processor Topics

- Transparent Performance Improvements
  - Pipelining, Caches
  - Superscalar, Out-of-Order, Branch Prediction, Speculation, ...
  - Covered in CS250A and others

- Explicit Performance Improvements
  - SIMD extensions, AES extensions, ...
  - ...

- Non-Performance Topics
  - Virtualization extensions, secure enclaves, transactional memory, ...
# Flynn Taxonomy (1966) Recap

<table>
<thead>
<tr>
<th>Instruction Stream</th>
<th>Data Stream</th>
<th>Single</th>
<th>Multi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>Single</td>
<td>SISD (Single-Core Processors)</td>
<td>SIMD (GPUs, Intel SSE/AVX extensions, ...)</td>
</tr>
<tr>
<td>Multi</td>
<td>Multi</td>
<td>MISD (Systolic Arrays, ...)</td>
<td>MIMD (VLIW, Parallel Computers)</td>
</tr>
</tbody>
</table>
Flynn Taxonomy Recap

- **Single-Instruction, Single-Data** (Single-Core Processors)
  - Data → Processing Unit

- **Multi-Instruction, Single-Data** (Systolic Arrays, ...)
  - Data → Processing Unit

- **Single-Instruction, Multi-Data** (GPUs, SIMD Extensions)
  - Data → Processing Unit

- **Multi-Instruction, Multi-Data** (Parallel Computers)
  - Data → Processing Unit
Intel SIMD Extensions

- New instructions, new registers
- Introduced in phases/groups of functionality
    - 128 bit width operations
    - 256 – 512 bit width operations
- F16C, and more to come?
Ice Lake Annotated Die (2019)

Die photo from Intel, annotation by Wikichip
Sandy Bridge Microarchitecture (2011)

e.g., “Port 5 pressure” when code uses too much shuffle operations
Skylake-X Microarchitecture (2019)
Intel SIMD Registers (AVX-512)

- XMM0 – XMM15
  - 128-bit registers
  - SSE
- YMM0 – YMM15
  - 256-bit registers
  - AVX, AVX2
- ZMM0 – ZMM31
  - 512-bit registers
  - AVX-512
SSE/AVX Data Types

Operation on 32 8-bit values in one instruction!
Complexity of AVX-512

The 12 levels of AVX512 in Intel processors
2017-06-01
according to Intel SDE 8.40

From @InstLatx64
Aside: Do I Have SIMD Capabilities?

- less /proc/cpuinfo

```
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat p
se36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm con
stant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmp
erf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx1
6 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f
16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd ibrs ibp
b stibp tpr_shadow vnumi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2
erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsave
es dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp flush_l1d
```
Processor Microarchitectural Effects on Power Efficiency

- The majority of power consumption of a CPU is not from the ALU
  - Cache management, data movement, decoding, and other infrastructure
  - Adding a few more ALUs should not impact power consumption

- Indeed, 4X performance via AVX does not add 4X power consumption
  - From i7 4770K measurements with matrix multiplication:
    - Idle: 40 W
    - Under load: 117 W
    - Under AVX load: 128 W
Compiler Automatic Vectorization

- In gcc, flags “-O3 -mavx -mavx2” attempts automatic vectorization
- Works pretty well for simple loops
- But not for anything complex
  - E.g., naïve bubblesort code not parallelized at all

```c
int a[256], b[256], c[256];
void foo () {
    for (int i=0; i<256; i++) a[i] = b[i] * c[i];
}
```

Generated using GCC explorer: https://gcc.godbolt.org/
Intel SIMD Intrinsics

- Use C functions instead of inline assembly to call AVX instructions
- Compiler manages registers, etc
- Intel Intrinsics Guide
  - One of my most-visited pages...

E.g.,
__m256 a, b, c;
__m256 d = _mm256_fmadd_ps(a, b, c); // d[i] = a[i]*b[i]+c[i] for i = 0 ... 7
## Data Types in AVX/AVX2

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>1~16 signed/unsigned integers</th>
<th>1~32 signed/unsigned integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>__m128</td>
<td>128-bit vector containing 4 floats</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__m128d</td>
<td>128-bit vector containing 2 doubles</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__m128i</td>
<td>128-bit vector containing integers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__m256</td>
<td>256-bit vector containing 8 floats</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__m256d</td>
<td>256-bit vector containing 4 doubles</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__m256i</td>
<td>256-bit vector containing integers</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

__m512 variants also for AVX-512
Intrinsic Naming Convention

- _mm<width>_[function]_[type]
  - E.g., _mm256_fmadd_ps: perform fmadd (fused multiply-add) on 256 bits of packed single-precision floating point values (8 of them)

<table>
<thead>
<tr>
<th>Width</th>
<th>Prefix</th>
<th>Type</th>
<th>Postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td><em>mm</em></td>
<td>Single precision</td>
<td>_ps</td>
</tr>
<tr>
<td>256</td>
<td><em>mm256</em></td>
<td>Double precision</td>
<td>_pd</td>
</tr>
<tr>
<td>512</td>
<td><em>mm512</em></td>
<td>Packed signed integer</td>
<td>_epiNNN (e.g., epi256)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Packed unsigned integer</td>
<td>_epuNNN (e.g., epu256)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scalar integer</td>
<td>_siNNN (e.g., si256)</td>
</tr>
</tbody>
</table>

Not all permutations exist! Check guide
Load/Store/Initialization Operations

- Initialization
  - `_mm256_setzero_ps/pd/epi32/...`
  - `_mm256_set_...`
  - `...`

- Load/Store: Variants for addresses aligned/unaligned by 256-bit
  - `_mm256_load_... _mm256_loadu_...`
  - `_mm256_store_... _mm256_storeu_...`

- And many more! (Masked read/write, strided reads, etc...)

E.g.,
```
__mm256d t = _mm256_load_pd(double const * mem); // loads 4 double values from mem to t
__mm256i v = _mm256_set_epi32(h,g,f,e,d,c,b,a); // loads 8 integer values to v
```
Vertical Vector Instructions

- **Add/Subtract/Multiply**
  - `_mm256_add/sub/mul/div_ps/pd/epi`
    - Mul only supported for epi32/epu32/ps/pd
    - Div only supported for ps/pd
    - Consult the guide!

- **Max/Min/GreaterThan/Equals**

- **Sqrt, Reciprocal, Shift, etc...**

- **FMA (Fused Multiply-Add)**
  - `(a*b)+c, -(a*b)-c, -(a*b)+c, and other permutations!`
  - Consult the guide!

- ...
Integer Multiplication Caveat

- Integer multiplication of two N bit values require 2N bits
- E.g., `__mm256_mul_epi32` and `__mm256_mul_epu32`
  - Only use the lower 4 32 bit values
  - Result has 4 64 bit values
- E.g., `__mm256_mullo_epi32` and `__mm256_mullo_epu32`
  - Uses all 8 32 bit values
  - Result has 8 truncated 32 bit values
- And more options!
  - Consult the guide...
Horizontal Vector Instructions

- **Horizontal add/subtraction**
  - Adds adjacent pairs of values
  - E.g., __m256d _mm256_hadd_pd (__m256d a, __m256d b)
Shuffling/Permutation

- **Within 128-bit lanes**
  - `_mm256_shuffle_ps/pd/_mm256_permute_ps/pd/_mm256_permutevar_ps/… (a,b, imm8)
  - `_mm256_permute2x128/4x64` : Uses 8 bit control
  - `_mm256_permutevar8x32/…` : Uses 256 bit control

- **Across 128-bit lanes**
  - `_mm256_permute2x128/4x64` : Uses 8 bit control
  - `_mm256_permutevar8x32/…` : Uses 256 bit control

- **Not all type permutations exist for each type, but variables can be cast back and forth between types**

Matt Scarpino, “Crunching Numbers with AVX and AVX2,” 2016
Blend

- Merges two vectors using a control
  - `_mm256_blend_...` : Uses 8 bit control
    - e.g., `_mm256_blend_epi32`
  - `_mm256_blendv_...` : Uses 256 bit control
    - e.g., `_mm256_blendv_epi8`
Alignr

- Right-shifts concatenated value of two registers, by byte
  - Often used to implement circular shift by using two same register inputs
  - _mm256_alignr_epi8 (a, b, count)

Example of 64-bit values being shifted by 8
Helper Instructions

- Cast
  - __mm256i <-> __mm256, etc...
  - Syntactic sugar -- does not spend cycles

- Convert
  - 4 floats <-> 4 doubles, etc...

- Movemask
  - __mm256 mask to -> int imm8

- And many more...
Our Current State Of Matrix Multiply: Blocked Multiplication

- Performance is best when working set fits into cache
  - But as shown, even 2048 x 2048 doesn’t fit in cache
  - -> 2048 * 2048 * 2048 elements read from memory for matrix B

- Solution: Divide and conquer! – Blocked matrix multiply
  - For block size 32 x 32 -> 2048 * 2048 * (2048/32) reads

\[
\begin{align*}
A1 & \quad A2 & \quad A3 \\
\times & \quad & \\
B1 & \quad B2 & \quad B3 \\
\end{align*}
\]

\[
C1 = C1 \text{ sub-matrix} = A1 \times B1 + A2 \times B2 + A3 \times B3 \ldots
\]
Blocked Matrix Multiply Evaluations

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Elapsed (s)</th>
<th>Normalized Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve</td>
<td>63.19</td>
<td>1</td>
</tr>
<tr>
<td>Transposed</td>
<td>10.39</td>
<td>6.08</td>
</tr>
<tr>
<td>Blocked (32)</td>
<td>7.35</td>
<td>8.60</td>
</tr>
</tbody>
</table>

- Bottlenecked by computation
- Bottlenecked by memory
- Bottlenecked by processor
- Bottlenecked by memory (Not scaling!)

- AVX Transposed reading from DRAM at 14.55 GB/s
  - $2048^3 \times 4 \text{ (Bytes)} / 2.20 \text{ (s)} = 14.55 \text{ GB/s}$
  - 1x DDR4 2400 MHz on machine -> 18.75 GB/s peak
  - Pretty close! Considering DRAM also used for other things (OS, etc)

- Multithreaded getting 32 GB/s effective bandwidth
  - Cache effects with small chunks
### Blocked Matrix Multiply Evaluations

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Elapsed (s)</th>
<th>Normalized Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve</td>
<td>63.19</td>
<td>1</td>
</tr>
<tr>
<td>Transposed</td>
<td>10.39</td>
<td>6.08</td>
</tr>
<tr>
<td>Blocked (32)</td>
<td>7.35</td>
<td>8.60</td>
</tr>
<tr>
<td>AVX Transposed</td>
<td>2.20</td>
<td>28.72</td>
</tr>
<tr>
<td>Blocked (32) AVX</td>
<td>1.50</td>
<td>42.13</td>
</tr>
<tr>
<td>4 Thread Blocked (32) AVX</td>
<td>1.09</td>
<td>57.97</td>
</tr>
</tbody>
</table>

- Using FMA SIMD, Cache-Oblivious AVX gets 19 GFLOPS
  - Theoretical peak is 3 GHz x 8 way SIMD == 24 GFLOPS... Close!

140x performance increase compared to the baseline!
Case Study: Sorting

- Important, fundamental application!
- Can be parallelized via divide-and-conquer
- How can SIMD help?
Reminder: Sorting Network

- Network structure for sorting fixed number of values
- Type of a “comparator network”
  - comparators perform compare-and-swap
- Easily pipelined and parallelized

Example 4-element sorting network
5 comparators, 3 cycle pipelined
Reminder: Sorting Network

- Simple to generate correct sorting networks, but optimal structures are not well-known

Source: Wikipedia (Sorting Network)

Some known optimal sorting networks
Typically, we are sorting more than one set of tuples
- If we have multiple tasks, we can have task-level parallelism – Optimized networks!
- Sort multiple tuples at the same time

We first need to transpose the 8 8-element variables
- Each variable has a value for each sorting network instance
- Non-SIMD works, or a string of unpackhi/unpacklo/blend
SIMD And Sorting Networks

- Some SIMD instructions have high throughput, but high latency
  - Data dependency between two consecutive max instructions can take 8 cycles on Skylake
  - If each parallel stage has less than 4 operations, pipeline may stall
    - Solution: Interleave two sets of parallel 8-tuple sorting
  - In reality, min/max means even for 4-tuples, pipeline is still filled

```c
__m256d _mm256_max_pd (__m256d a, __m256d b)
```

Performance Table:

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Latency</th>
<th>Throughput (CPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skylake</td>
<td>4</td>
<td>0.5</td>
</tr>
<tr>
<td>Broadwell</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Haswell</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Ivy Bridge</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Source: Instrinsics guide
The Two Register Merge

- Sort units of two pre-sorted registers, $K$ elements
  - $\text{minv} = A$, $\text{maxv} = B$

  - // Repeat $K$ times
    - $\text{minv} = \text{min}(\text{minv}, \text{maxv})$
    - $\text{maxv} = \text{max}(\text{minv}, \text{maxv})$
    - // circular shift one value down
      - $\text{minv} = \text{alignr}(\text{minv}, \text{minv}, \text{sizeof(int)})$

---

SIMD And Merge Sort

- Hierarchically merged sorted subsections
- Using the SIMD merger for sorting
  - `vector_merge` is the two-register sorter from before

```c
aPos = bPos = outPos = 0;
vMin = va[aPos++];
vMax = vb[bPos++];
while (aPos < aEnd && bPos < bEnd) {
    /* merge vMin and vMax */
    vector_merge(vMin, vMax);
    /* store the smaller vector as output*/
    vMergedArray[outPos++] = vMin;
    /* load next vector and advance pointer */
    /* a[aPos*4] is first element of va[aPos] */
    /* and b[bPos*4] is that of vb[bPos] */
    if (a[aPos*4] < b[bPos*4])
        vMin = va[aPos++];
    else
        vMin = vb[bPos++];
}
```

Topic Under Active Research!

- Papers being written about...
  - Architecture-optimized matrix transposition
  - Register-level sorting algorithm
  - Merge-sort
  - ... and more!

- Good find can accelerate your application kernel Nx
Questions?