Flynn taxonomy

Single-Instruction Single-Data (Single-Core Processors)

Multi-Instruction Single-Data (Systolic Arrays,....)

Single-Instruction Multi-Data (GPUs, Intel SIMD Extensions)

Multi-Instruction Multi-Data (Parallel Processors)
Modern Processor Topics - Performance

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

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

SIMD operations

- Single ISA instruction performs same computation on multiple data
- Typically implemented with special, wider registers
- Example operation:
  - Load 32 bytes from memory to special register X
  - Load 32 bytes from memory to special register Y
  - Perform addition between each 4-byte value in X and each 4 byte value in Y
  - Store the four results in special register Z
  - Store Z to memory

- RISC-V SIMD extensions (P) still being worked on (2023)
- Vector extensions (V) ratified
Example: Intel SIMD Extensions

- More transistors (Moore’s law) but no faster clock, no more ILP...
  - More capabilities per processor has to be explicit!

- New instructions, new registers
  - Must be used explicitly by programmer or compiler!

- Introduced in phases/groups of functionality
    - 128 bit width operations
    - 256 – 512 bit width operations
  - F16C, and more to come?
Skylake-X Microarchitecture (2019)
Apple M1 Microarchitecture (2020)

Anandtech
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 pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsaves xsavec xsaveopt xgetbv1 xsaveas dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp flush_l1d
```
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!
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:
    - 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
  - 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.,
  ```c
  __m256 a, b, c;
  __m256 d = _mm256_fmadd_ps(a, b, c); // d[i] = a[i]*b[i]+c[i] for i = 0 ...7
  ```
Intrinsic Naming Convention

- _mm<width>_[function]_[type]
  - E.g., _mm256_fmadd_ps: perform fmadd (floating point 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
Example: Vertical Vector Instructions

- Add/Subtract/Multiply
  - `_mm256_add/sub/mul/div_ps/pd/epi`
    - Mul only supported for epi32/epi32/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!

- ...

```c
__m256 a, b, c;
__m256 d = _mm256_fmadd_pd(a, b, c);
```
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!
Case Study: Matrix Multiply

Branch:
Boser & Katz,
“CS61C: Great Ideas In Computer Architecture”
Lecture 18 – Parallel Processing – SIMD
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!
Example AVX matrix mult

```c
__m256 z = _mm256_set1_ps(0);
for ( int k = 0; k < matrix_size/8; k++ ){
    __m256 x = _mm256_loadu_ps(&a[i*matrix_size+k*8]);
    __m256 y = _mm256_loadu_ps(&b[j*matrix_size+k*8]);
    z = _mm256_fmadd_ps(x,y,z);
}
```