CS250P: Computer Systems Architecture
Explicit Parallelism

Sang-Woo Jun
Fall 2023

Large amount of material adapted from MIT 6.004, “Computation Structures”, Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”, and CS 152 Slides by Isaac Scherson
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) is still being worked on (as of 2021)
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?

Performance matters, “AVX-512 Mask Registers, Again.” 2020
Aside: Do I Have SIMD Capabilities?

- less /proc/cpuinfo

```bash
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 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 pcds syscall tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssmbd 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 xsavec xsavec xgetbv1 xsaveas dtherm ida arat pni pts hwp hwp_notify hwp_act_window hwp_epp flush_l1d flush_l1i popcron popcnt tscMapper seclvl1thm eleven amd prefect vtes64p vtes64e pxrsvts smc tsc_adjust cplcpu
```
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!
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.,

```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>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>_mm_</td>
</tr>
<tr>
<td>256</td>
<td>_mm256_</td>
</tr>
<tr>
<td>512</td>
<td>_mm512_</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single precision</td>
<td>_ps</td>
</tr>
<tr>
<td>Double precision</td>
<td>_pd</td>
</tr>
<tr>
<td>Packed signed integer</td>
<td>_epiNNN (e.g., epi256)</td>
</tr>
<tr>
<td>Packed unsigned integer</td>
<td>_epuNNN (e.g., epu256)</td>
</tr>
<tr>
<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/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!

- **...**

```c
__m256 a, b, c;
__m256 d = _mm256_fmadd_pd(a, b, c);
```
Horizontal Vector Instructions

- Horizontal add/subtraction
  - Adds adjacent pairs of values
  - E.g., `__m256d_mm256_hadd_pd(__m256d a, __m256d b)`

![Diagram showing horizontal addition]
Shuffling/Permutation

- **Within 128-bit lanes**
  - _mm256_shuffle_ps/pd/... (a,b, imm8)
  - _mm256_permute_ps/pd
  - _mm256_permutevar_ps/...

- **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...
Case Study: Matrix Multiplication

- Remember simply transposing matrix B brought 6x performance
  - At that point, we are bottlenecked by single-thread processing performance
  - Adding SIMD gets us more!
  - After this we are again bottlenecked by memory, but that is for another time

- $A \times B$ VS $63.19 \text{ seconds}$

- $A \times B^T$ 2.20 seconds (29x performance!)
Case Study: Sorting

- Important, fundamental application!
- Can be parallelized via divide-and-conquer
- How can SIMD help?
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
    - $\text{minv} = \text{alignr}($minv, minv, 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

```
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
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