Modern Computer Architecture

Lecture 7
SIMD Extensions
MMX/SSE/AVX/AVX2
ARM Advanced SIMD Extensions
Outline – Lecture 7

• Flynn’s Taxonomy
• SIMD Extensions
  – x86 world: MMX/ISSE/SSE/AVX/AVX2
  – AMD 3D-Now!, IBM/Sony Cell, IBM AltiVec, etc.
  – ARM VFP (vector floating point), Advanced SIMD
Flynn’s Taxonomy

SISD

Instruction Pool

Data Pool

PE

MISD

Instruction Pool

Data Pool

PE

PE

SIMD

Instruction Pool

Data Pool

PE

PE

PE

MIMD

Instruction Pool

Data Pool

PE

PE

PE

PE

PE

PE
Why SIMD and not Super-Scalar?

- Super-Scalar can have hundreds of instructions “in-flight”
- Issue up to 10 instructions per clock cycle
- Sources of performance limitations:
  - number of computation resources
  - branch misprediction
  - cache misses
- Actual IPC rarely exceeds 2
- Control overhead consumes majority of resources

Solution: SIMD instructions
- Perform several operations with one instruction
for (i=0; i<n; i++)
    c[i] = a[i] + b[i];

\[
c[0:n-1] =_{\text{vector}} a[0:n-1] +_{\text{vector}} b[0:n-1];
\]

for (i=0; i<n; i+=s)
    c[i:i+s-1] =_{\text{SIMD}} a[i:i+s-1] +_{\text{SIMD}} b[i:i+s-1];

Randy Allen, Ken Kennedy: Optimizing Compilers for Modern Architectures

- Cray-1: s=64
- Intel: s=4/8/16

Vectorization

Strip-Mining
Data Types in Vector Processing

- **Predicate**
  - Bits
  - Control Flow
  - True/False Decisions

- **Scalar**
  - Words
  - Individual values
  - Numbers/bit vectors

- **Vector**
  - Cache Lines
  - Array values
  - Number vectors
• after the fact extensions
• idea: use architectural register to store multiple values
• not oriented at PEs, but on data width:
  – “SWP” Sub-Word Parallelism

• for example: 64-bit register can hold
  – 1x double precision float
  – 2x single precision float
  – 4x 16-bit integer
  – 8x 8-bit integer

• First Applications:
  – 3D Graphics (Geometry&Lighting)
  – DVD Playback
• x86 MMX: 64-bit register can hold
  – 1x double precision float
  – 2x single precision float
  – 4x 16-bit integer
  – 8x 8-bit integer

<table>
<thead>
<tr>
<th></th>
<th>64</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

• x86 SSE: 128-bit registers xmm0-xmm7
  – 2x double precision float
  – 4x single precision float
  – 8x 16-bit integer
  – 16x 8-bit integer

<table>
<thead>
<tr>
<th></th>
<th>64</th>
<th>32</th>
<th>64</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

• x86 AVX/AVX2: 256-bit registers ymm0-ymm15
  – 4x double precision float
  – 8x single precision float

<table>
<thead>
<tr>
<th></th>
<th>64</th>
<th>64</th>
<th>64</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>
Historical Development

- **MMX extensions (1996)**
  - integer calculations
  - alternate usage for 8x 80-bit floating point registers: \( mm0\ldots mm7 \)
  - no operating system support required for context switch
  - only 8 vector registers (\( \rightarrow \) x86-64, AVX: 16)
  - 2-address form (\( \rightarrow \) AVX: 3-address form)
  - used for image processing

- **SSE (1999)**
  - integer and floating point
  - 8x 128-bit registers \( xmm0\ldots xmm7 \)

- **AVX (2011)**
  - 16x 256-bit wide registers: \( ymm0\ldots ymm15 \)
  - 3-address form

- **AVX2 (2013)**
  - fused multiply-accumulate (FMA)
  - gather support

- **Larrabee / Xeon Phi / Knight (2012)**
  - 32x 512-bit wide registers: \( zmm0\ldots zmm31 \)
Examples of Instructions

- **MOVUPS, MOVAPS**
  - move instruction
  - data types: PS $\Rightarrow$ “packed single precision float”
  - U $\Rightarrow$ unaligned memory address (penalty), A $\Rightarrow$ aligned memory address

- **ADDPS (“packed single”)**
  - Adds two registers containing 4x single precision floating point values

- **ADDSS (“scalar single”)**
  - Adds two single precision floating point values at the bottom position of the register

- **MULPS**
  - Multiples two registers containing 4x single precision floating point values
• Conditional Move
  avoids branches

• Scalar: FCMOV
  uses flags (comparison operations)

• SIMD: CMPSS CMPPS CMPPD
  – set bits masks with 0s and 1s
  – use wide bitwise AND/OR/NOT
How-To use SIMD on PC

- Assembly Programming
  
  ```assembly
  movaps xmm0, $a
  movaps xmm1, $b
  addps xmm0, xmm1
  movaps $c, cmm0
  ```

- Intrinsics
  
  ```c
  __m128 var0, var1, var3;
  var0 = _mm_load_ps(a);
  var1 = _mm_load_ps(b);
  var3 = _mm_add_ps(var0, var1);
  _mm_store_ps(c, var3);
  ```

- Vectorization
  
  ```c
  for (i = 0; i < 4; i++)
      c[i] = a[i] + b[i];
  ```
for (i=0; i<N; i++)
    for (j=0; j<N; j++) {
        m3[i][j] = 0.0;
        for (k=0; k<M; k++)
            m3[i][j] += m1[i][k] * m2[k][j];
    }

..B2.24:
    movups m2(%rdx,%rdi,4), %xmm1
    movups 16+m2(%rdx,%rdi,4), %xmm2
    mulps %xmm0, %xmm1
    mulps %xmm0, %xmm2
    addps m3(%rcx,%rdi,4), %xmm1
    addps 16+m3(%rcx,%rdi,4), %xmm2
    movaps %xmm1, m3(%rcx,%rdi,4)
    movaps %xmm2, 16+m3(%rcx,%rdi,4)
    addq $8, %rdi
    cmpq %rsi, %rdi
    jb ..B2.24
Data Arrangement Operations

- Insert Element into Vector
- Extract Element from Vector
- Shift Vector Elements

→ Load and Store into Memory, pointer arithmetic

- utilize infrastructure from load/store queue
- inefficient for large vectors
Shuffle Operations

- Use Case: Butterfly for FFT
- Matrix Transpose
- Conversion from AOS to SOA and vice versa
  - **AOS**: array of structures
    
    ```c
    struct { float x, y, z } [N];
    ```
  - **SOA**: structure of arrays
    
    ```c
    struct { float x[N], y[N], z[N] };
    ```

- 1x shuffle to broadcast
- ➞ 3-4x shuffle to transpose
Shuffle: Broadcast

\[
\text{shufps } \text{xmm0, xmm0, 0h}
\]

\[
\begin{array}{cccc}
00 & 00 & 00 & 00 \\
\end{array}
\]

\[
\text{shufps } \text{xmm0, xmm0, 55h}
\]

\[
\begin{array}{cccc}
01 & 01 & 01 & 01 \\
\end{array}
\]
Reverse Vector

```assembly
shufps xmm0, xmm0, 1bh
```

```
| 00 | 01 | 10 | 11 |
```

Pick Every 2nd Element

```assembly
shufps xmm0, xmm1, 88h
```

```
| 10 | 00 | 10 | 00 |
```
• TBD
AVX/AVX2

- more registers 8 → 16
- 3-address code
- AES encryption extensions
- from 128 to 256 bits: __m128 → __m256
- Most instructions get ‘V’ prefix: e.g. ADDPS → VADDPs
- Better data arrangement operations: blend, permute

AVX2

- gather instruction
- FMA (fused-multiply-add): retain high-precision intermediate result
  - (MAC multiply-accumulate: intermediate rounding)
- more registers $16 \rightarrow 32$ (zmm0 … zmm31)
- from 256 to 512 bits: `__m256` $\rightarrow$ `__m512`
- vector mask registers k0-k7

- 512-bit registers zmm0-zmm31
  - 8x double precision float
  - 16x single precision float

<table>
<thead>
<tr>
<th>64</th>
<th>64</th>
<th>64</th>
<th>64</th>
<th>64</th>
<th>64</th>
<th>64</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>
Performance Analysis

• Intel: Amplifier, V-Tune, XE Parallel Studio, Intel Compiler

• AMD: Code Analyst, GCC, Open64 Compiler

• NVIDIA: CUDA
• http://infocenter.arm.com/
• http://weinberg.userweb.mwn.de/mic/micworkshop-micprogramming.pdf