Lecture 21: Data Level Parallelism
-- SIMD ISA Extensions for Multimedia and Roofline Performance Model

CSCE 513 Computer Architecture

Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
https://passlab.github.io/CSCE513
Topics for Data Level Parallelism (DLP)

- Parallelism (centered around … )
  - Instruction Level Parallelism
  - Data Level Parallelism
  - Thread Level Parallelism

- DLP Introduction and Vector Architecture
  - 4.1, 4.2

- SIMD Instruction Set Extensions for Multimedia
  - 4.3

- Graphical Processing Units (GPU)
  - 4.4

- GPU and Loop-Level Parallelism and Others
  - 4.4, 4.5
SIMD Instruction Set extension for Multimedia
Textbook: CAQA 4.3
What is Multimedia

- Multimedia is a combination of text, graphic, sound, animation, and video that is delivered interactively to the user by electronic or digitally manipulated means.

<table>
<thead>
<tr>
<th>Medium</th>
<th>Elements</th>
<th>Time-dependence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>Printable characters</td>
<td>No</td>
</tr>
<tr>
<td>Graphic</td>
<td>Vectors, regions</td>
<td>No</td>
</tr>
<tr>
<td>Image</td>
<td>Pixels</td>
<td>No</td>
</tr>
<tr>
<td>Audio</td>
<td>Sound, Volume</td>
<td>Yes</td>
</tr>
<tr>
<td>Video</td>
<td>Raster images, graphics</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Videos contains frame (images)

https://en.wikipedia.org/wiki/Multimedia
Image Format and Processing

- **Pixels**
  - Images are matrix of pixels

- **Binary images**
  - Each pixel is either 0 or 1
Image Format and Processing

- **Pixels**
  - Images are matrix of pixels

- **Grayscale images**
  - Each pixel value normally range from 0 (black) to 255 (white)
  - 8 bits per pixel
Image Format and Processing

- **Pixels**
  - Images are matrix of pixels

- **Color images**
  - Each pixel has three/four values (4 bits or 8 bits each) each representing a color scale
Image Processing

- Mathematical operations by using any form of signal processing
  - Changing pixel values by matrix operations

![Diagram of convolution kernel (emboss)](image)

Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.

\[
\begin{align*}
(4 \times 0) \\
(0 \times 0) \\
(0 \times 0) \\
(0 \times 0) \\
(0 \times 1) \\
(0 \times 1) \\
(0 \times 0) \\
(0 \times 1) \\
\hline
+ (-4 \times 2) \\
\hline
-8
\end{align*}
\]

Blur the source horizontally
Blur the blur vertically
Result

Image taken from ATI's presentation
# Image Processing: The major of the filter matrix

- [http://lodev.org/cgtutor/filtering.html](http://lodev.org/cgtutor/filtering.html)

## Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.

### Identity

<table>
<thead>
<tr>
<th><img src="image" alt="Identity" /></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Identity" /></td>
</tr>
</tbody>
</table>

\[
\begin{bmatrix}
0 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
\]

### Edge detection

<table>
<thead>
<tr>
<th><img src="image" alt="Edge detection" /></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Edge detection" /></td>
</tr>
</tbody>
</table>

\[
\begin{bmatrix}
1 & 0 & -1 \\
0 & 0 & 0 \\
-1 & 0 & 1 \\
\end{bmatrix}
\]

### Sharpen

<table>
<thead>
<tr>
<th><img src="image" alt="Sharpen" /></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Sharpen" /></td>
</tr>
</tbody>
</table>

\[
\begin{bmatrix}
0 & -1 & 0 \\
-1 & 5 & -1 \\
0 & -1 & 0 \\
\end{bmatrix}
\]

### Box blur (normalized)

<table>
<thead>
<tr>
<th><img src="image" alt="Box blur" /></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Box blur" /></td>
</tr>
</tbody>
</table>

\[
\frac{1}{9}
\begin{bmatrix}
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 1 \\
\end{bmatrix}
\]

### Gaussian blur (approximation)

<table>
<thead>
<tr>
<th><img src="image" alt="Gaussian blur" /></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Gaussian blur" /></td>
</tr>
</tbody>
</table>

\[
\frac{1}{16}
\begin{bmatrix}
1 & 2 & 1 \\
2 & 4 & 2 \\
1 & 2 & 1 \\
\end{bmatrix}
\]
Image Data Format and Processing for SIMD Architecture

- Data element
  - 4, 8, 16 bits (small)

- Same operations applied to every element (pixel)
  - Perfect for data-level parallelism

Can fit multiple pixels in a regular scalar register

- E.g. for 8 bit pixel, a 64-bit register can take 8 of them
Multimedia Extensions (aka SIMD extensions) to Scalar ISA

- Very short vectors added to existing ISAs for microprocessors
- Use existing 64-bit registers split into 2x32b or 4x16b or 8x8b
  - Lincoln Labs TX-2 from 1957 had 36b datapath split into 2x18b or 4x9b
  - Newer designs have wider registers
    » 128b for PowerPC Altivec, Intel SSE2/3/4
    » 256b for Intel AVX
- Single instruction operates on all elements within register

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

4x16b adds

![Diagram of 4x16b adds](image-url)
A Scalar FU to A Multi-Lane SIMD Unit

- Adder
  - Partitioning the carry chains

![Diagram of a 64-bit lookahead carry unit with four 16-bit LCUs]

<table>
<thead>
<tr>
<th>Instruction category</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned add/subtract</td>
<td>Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit</td>
</tr>
<tr>
<td>Maximum/minimum</td>
<td>Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit</td>
</tr>
<tr>
<td>Average</td>
<td>Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit</td>
</tr>
<tr>
<td>Shift right/left</td>
<td>Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit</td>
</tr>
<tr>
<td>Floating point</td>
<td>Sixteen 16-bit, eight 32-bit, four 64-bit, or two 128-bit</td>
</tr>
</tbody>
</table>

**Figure 4.8 Summary of typical SIMD multimedia support for 256-bit-wide operations.** Note that the IEEE 754-2008 floating-point standard added half-precision (16-bit) and quad-precision (128-bit) floating-point operations.
MMX SIMD Extensions to X86

- MMX instructions added in 1996
  - Repurposed the 64-bit floating-point registers to perform 8 8-bit operations or 4 16-bit operations simultaneously.
  - MMX reused the floating-point data transfer instructions to access memory.
  - Parallel MAX and MIN operations, a wide variety of masking and conditional instructions, DSP operations, etc.

- Claim: overall speedup 1.5 to 2X for 2D/3D graphics, audio, video, speech, comm., ...
  - use in drivers or added to library routines; no compiler
MMX Instructions

- Move 32b, 64b
- Add, Subtract in parallel: 8 8b, 4 16b, 2 32b
  - opt. signed/unsigned saturate (set to max) if overflow
- Shifts (sll, srl, sra), And, And Not, Or, Xor
  in parallel: 8 8b, 4 16b, 2 32b
- Multiply, Multiply-Add in parallel: 4 16b
- Compare =, > in parallel: 8 8b, 4 16b, 2 32b
  - sets field to 0s (false) or 1s (true); removes branches
- Pack/Unpack
  - Convert 32b<---> 16b, 16b <--> 8b
  - Pack saturates (set to max) if number is too large
SSE/SSE2/SSE3 SIMD Extensions to X86

- Streaming SIMD Extensions (SSE) successor in 1999
  - Added separate 128-bit registers that were 128 bits wide
    » 16 8-bit operations, 8 16-bit operations, or 4 32-bit operations.
    » Also perform parallel single-precision FP arithmetic.
  - Separate data transfer instructions.
    » increased the peak FP performance of the x86 computers.
  - Each generation also added ad hoc instructions to accelerate specific multimedia functions.
AVX SIMD Extensions for X86

- Advanced Vector Extensions (AVX), added in 2010
- Doubles the width of the registers to 256 bits – double the number of operations on all narrower data types. Figure 4.9 shows AVX instructions useful for double-precision floating-point computations.
- AVX includes preparations to extend to 512 or 1024 bits in future generations of the architecture.

<table>
<thead>
<tr>
<th>AVX Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VADDPD</td>
<td>Add four packed double-precision operands</td>
</tr>
<tr>
<td>VSUBPD</td>
<td>Subtract four packed double-precision operands</td>
</tr>
<tr>
<td>VMULPD</td>
<td>Multiply four packed double-precision operands</td>
</tr>
<tr>
<td>VDIVPD</td>
<td>Divide four packed double-precision operands</td>
</tr>
<tr>
<td>VFMAADDPD</td>
<td>Multiply and add four packed double-precision operands</td>
</tr>
<tr>
<td>VFMSUBPD</td>
<td>Multiply and subtract four packed double-precision operands</td>
</tr>
<tr>
<td>VCMPxx</td>
<td>Compare four packed double-precision operands for EQ, NEQ, LT, LE, GT, GE, …</td>
</tr>
<tr>
<td>VMOVAPD</td>
<td>Move aligned four packed double-precision operands</td>
</tr>
<tr>
<td>VBBROADCASTSD</td>
<td>Broadcast one double-precision operand to four locations in a 256-bit register</td>
</tr>
</tbody>
</table>
DAXPY

```plaintext
double a, X[], Y[]; // 8-byte
for (i=0; i<32; i++)
  Y[i] = a* X[i] + Y[i];
```

- **256-bit SIMD exts to RISC-V ➔ RVP**
  - 4 double FP

- **RV64G: 258 insts**
- **SIMD RVP: 67 insts**
  - 8 Loop iterations
  - 4× reduction
- **RV64V: 8 instrs**
  - 30× reduction

```plaintext
vsetdcfg  4*FP64       # Enable 4 DP FP vregs
fld   f0,a       # Load scalar a
vld   v0,x5      # Load vector X
vmul  v1,v0,f0   # Vector-scalar mult
vld   v2,x6      # Load vector Y
vadd  v3,v1,v2   # Vector-vector add
vst   v3,x6      # Store the sum
vdisable
```

```plaintext
fld   f0,a       # Load scalar a
splat.4D f0,f0     # Make 4 copies of a
addi  x28,x5,#256 # Last address to load
Loop: fld.4D f1,0(x5) # Load X[i]...X[i+3]
      fmul.4D f1,f1,f0 # a×X[i]...a×X[i+3]
      fld.4D f2,0(x6) # Load Y[i]...Y[i+3]
      fadd.4D f2,f2,f1 # a×X[i]+Y[i]...
      fsd.4D f2,0(x6) # Store Y[i]...Y[i+3]
      addi  x5,x5,#32 # Increment index to X
      addi  x6,x6,#32 # Increment index to Y
      bne   x28,x5,Loop # Check if done
```
Multimedia Extensions versus Vectors

- Limited instruction set:
  - no vector length control
  - no strided load/store or scatter/gather
  - unit-stride loads must be aligned to 64/128-bit boundary

- Limited vector register length:
  - requires superscalar dispatch to keep multiply/add/load units busy
  - loop unrolling to hide latencies increases register pressure

- Trend towards fuller vector support in microprocessors
  - Better support for misaligned memory accesses
  - Support of double-precision (64-bit floating-point)
  - New Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b)
The easiest way to use these instructions has been through libraries or by writing in assembly language.
- The ad hoc nature of the SIMD multimedia extensions,

Recent extensions have become more regular
- Compilers are starting to produce SIMD instructions automatically.
  » Advanced compilers today can generate SIMD FP instructions to deliver much higher performance for scientific codes.
  » Memory alignment is still an important factor for performance
Why are Multimedia SIMD Extensions so Popular

- Cost little to add to the standard arithmetic unit and they were easy to implement.
- Require little extra state compared to vector architectures, which is always a concern for context switch times.
- Does not require a lot of memory bandwidth to support as what a vector architecture requires.
- Others regarding to the virtual memory and cache that make SIMD extensions less challenging than vector architecture.

The state of the art is that we are putting a full or advanced vector capability to multi/manycore CPUs, and Manycore GPUs
State of the Art: Intel Xeon Phi Manycore Vector Capability

- Intel Xeon Phi Knight Corner, 2012, ~60 cores, 4-way SMT
- Intel Xeon Phi Knight Landing, 2016, ~60 cores, 4-way SMT and HBM

- [Link to Primeur Magazine Article](http://primeurmagazine.com/repository/PrimeurMagazine-AE-PR-12-14-32.pdf)

```c
#define N 1000000
float x[N][N], y[N][N];
#pragma omp parallel
{
    #pragma omp for
    for (int i=0; i<N; i++) {
        #pragma omp simd safelen(18)
        for (int j=18; j<N-18; j++) {
            x[i][j] = x[i][j-18] + sinf(y[i][j]);
            y[i][j] = y[i][j+18] + cosf(x[i][j]);
        }
    }
}
```
The Picture I drew on the blackBoard

4 threads on four cores
each core can do SIMD execution
N = 100

OMP parallel

OMP for
loop distribution 0-24 25-49 50-74 75-99

OMP SIMD

All for cores do M2MD parallel

each core does SIMD execution
State of the Art: ARM Scalable Vector Extensions (SVE)

- Announced in August 2016

- Beyond vector architecture we learned
  - Vector loop, predict and speculation
  - Vector Length Agnostic (VLA) programming
  - Check the slide
The Roofline Visual Performance Model

- Self-study if you are interested: two pages of textbook
  - Useful, simple and interesting

- More materials:
  - Slides: https://crd.lbl.gov/ assets/pubs_presos/parlab08-roofline-talk.pdf
  - Paper: https://people.eecs.berkeley.edu/~waterman/papers/roofline.pdf
  - Website: https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/