Computer Architecture: SIMD and GPUs (Part I)

Prof. Onur Mutlu
Carnegie Mellon University
A Note on This Lecture

- These slides are partly from 18-447 Spring 2013, Computer Architecture, Lecture 15: Dataflow and SIMD

- Video of full lecture from 447:
  - http://www.youtube.com/watch?v=f-XL4BNRoBA

- Video of the part related to only SIMD and GPUs:
  - https://youtube.googleapis.com/v/f-XL4BNRoBA%26start=4980
Vector Processing: Exploiting Regular (Data) Parallelism
Flynn’s Taxonomy of Computers


- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- **MISD**: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor
Data Parallelism

- Concurrency arises from performing the same operations on different pieces of data
  - Single instruction multiple data (SIMD)
  - E.g., dot product of two vectors

- Contrast with data flow
  - Concurrency arises from executing different operations in parallel (in a data driven manner)

- Contrast with thread ("control") parallelism
  - Concurrency arises from executing different threads of control in parallel

- SIMD exploits instruction-level parallelism
  - Multiple instructions concurrent: instructions happen to be the same
SIMD Processing

- Single instruction operates on multiple data elements
  - In time or in space
- Multiple processing elements

- Time-space duality
  - **Array processor**: Instruction operates on multiple data elements at the same time
  - **Vector processor**: Instruction operates on multiple data elements in consecutive time steps
Array vs. Vector Processors

Instruction Stream

| LD       | VR ← A[3:0] |
| ADD     | VR ← VR, 1  |
| MUL     | VR ← VR, 2  |
| ST      | A[3:0] ← VR |

Time

Array Processor

- Same op @ same time
- Different ops @ same space

Vector Processor

- Different ops @ time
- Same op @ space

Space
SIMD Array Processing vs. VLIW

- VLIW
SIMD Array Processing vs. VLIW

- Array processor

![Diagram of SIMD Array Processing vs. VLIW](image-url)
Vector Processors

- A vector is a one-dimensional array of numbers
- Many scientific/commercial programs use vectors
  ```
  for (i = 0; i<=49; i++)
  C[i] = (A[i] + B[i]) / 2
  ```
- A vector processor is one whose instructions operate on vectors rather than scalar (single data) values
- Basic requirements
  - Need to load/store vectors → vector registers (contain vectors)
  - Need to operate on vectors of different lengths → vector length register (VLEN)
  - Elements of a vector might be stored apart from each other in memory → vector stride register (VSTR)
    - Stride: distance between two elements of a vector
Vector Processors (II)

- A vector instruction performs an operation on each element in consecutive cycles
  - Vector functional units are pipelined
  - Each pipeline stage operates on a different data element

- Vector instructions allow deeper pipelines
  - No intra-vector dependencies → no hardware interlocking within a vector
  - No control flow within a vector
  - Known stride allows prefetching of vectors into cache/memory
Vector Processor Advantages

+ No dependencies within a vector
  - Pipelining, parallelization work well
  - Can have very deep pipelines, no dependencies!

+ Each instruction generates a lot of work
  - Reduces instruction fetch bandwidth

+ Highly regular memory access pattern
  - Interleaving multiple banks for higher memory bandwidth
  - Prefetching

+ No need to explicitly code loops
  - Fewer branches in the instruction sequence
Vector Processor Disadvantages

-- Works (only) if parallelism is regular (data/SIMD parallelism)
  ++ Vector operations
-- Very inefficient if parallelism is irregular
  -- How about searching for a key in a linked list?

To program a vector machine, the compiler or hand coder must make the data structures in the code fit nearly exactly the regular structure built into the hardware. That’s hard to do in first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated programmer who knows the hardware and often the subtleties of the application area. Often the rewriting is

Vector Processor Limitations

-- Memory (bandwidth) can easily become a bottleneck, especially if

1. compute/memory operation balance is not maintained
2. data is not mapped appropriately to memory banks
We did not cover the following slides in lecture. These are for your preparation for the next lecture.
Vector Registers

- Each vector data register holds \( N \) \( M \)-bit values
- Vector control registers: VLEN, VSTR, VMASK
- Vector Mask Register (VMASK)
  - Indicates which elements of vector to operate on
  - Set by vector test instructions
    - e.g., VMASK\([i]\) = \((V_k[i] == 0)\)
- Maximum VLEN can be \( N \)
  - Maximum number of elements stored in a vector register
Vector Functional Units

- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent

\[ V_3 \leftarrow v_1 \times v_2 \]

*Six stage multiply pipeline*
Vector Machine Organization (CRAY-1)

- CRAY-1

- Scalar and vector modes
- 8 64-element vector registers
- 64 bits per element
- 16 memory banks
- 8 64-bit scalar registers
- 8 24-bit address registers
Memory Banking

- Example: 16 banks; can start one bank access per cycle
- Bank latency: 11 cycles
- Can sustain 16 parallel accesses if they go to different banks
Vector Memory System

Vector Registers

Address Generator

Memory Banks

Base Stride

Slide credit: Krste Asanovic
Scalar Code Example

- For I = 0 to 49
  - C[i] = (A[i] + B[i]) / 2

- Scalar code
  MOV R0 = 50 1
  MOV R1 = A 1
  MOV R2 = B 1
  MOV R3 = C 1
  X: LD R4 = MEM[R1++] 11 ;autoincrement addressing
  LD R5 = MEM[R2++] 11
  ADD R6 = R4 + R5 4
  SHFR R7 = R6 >> 1 1
  ST MEM[R3++] = R7 11
  DECBNZ --R0, X 2 ;decrement and branch if NZ

304 dynamic instructions
Scalar Code Execution Time

- Scalar execution time on an in-order processor with 1 bank
  - First two loads in the loop cannot be pipelined: 2*11 cycles
  - 4 + 50*40 = 2004 cycles

- Scalar execution time on an in-order processor with 16 banks (word-interleaved)
  - First two loads in the loop can be pipelined
  - 4 + 50*30 = 1504 cycles

- Why 16 banks?
  - 11 cycle memory access latency
  - Having 16 (>11) banks ensures there are enough banks to overlap enough memory operations to cover memory latency
Vectorizable Loops

- A loop is **vectorizable** if each iteration is independent of any other.

- For $I = 0$ to $49$
  - $C[i] = (A[i] + B[i]) / 2$

- Vectorized loop:
  - MOV1 VLEN = 50
  - MOV1 VSTR = 1
  - VLD V0 = A
  - VLD V1 = B
  - VADD V2 = V0 + V1
  - VSHFR V3 = V2 >> 1
  - VST C = V3

7 dynamic instructions
Vector Code Performance

- No chaining
  - i.e., output of a vector functional unit cannot be used as the input of another (i.e., no vector data forwarding)
- One memory port (one address generator)
- 16 memory banks (word-interleaved)

= 49

= 49

V0 = A[0..49] | V1 = B[0..49] | ADD | SHIFT | STORE |

- 285 cycles
Vector Chaining

- **Vector chaining**: Data forwarding from one vector functional unit to another

Slide credit: Krste Asanovic
Vector Code Performance - Chaining

- **Vector chaining**: Data forwarding from one vector functional unit to another

- 182 cycles

These two VLDs cannot be pipelined. **WHY?**

VDL and VST cannot be pipelined. **WHY?**

**Strict assumption**: Each memory bank has a single port (memory bandwidth bottleneck)
Vector Code Performance – Multiple Memory Ports

- Chaining and 2 load ports, 1 store port in each bank
- 79 cycles
Questions (I)

- What if # data elements > # elements in a vector register?
  - Need to break loops so that each iteration operates on # elements in a vector register
    - E.g., 527 data elements, 64-element VREGs
    - 8 iterations where VLEN = 64
    - 1 iteration where VLEN = 15 (need to change value of VLEN)
  - Called **vector stripmining**

- What if vector data is not stored in a strided fashion in memory? (irregular memory access to a vector)
  - Use indirection to combine elements into vector registers
  - Called **scatter/gather operations**
Want to vectorize loops with indirect accesses:

```c
for (i=0; i<N; i++)
    A[i] = B[i] + C[D[i]]
```

Indexed load instruction (*Gather*)

```assembly
LV vD, rD       # Load indices in D vector
LVI vC, rC, vD  # Load indirect from rC base
LV vB, rB       # Load B vector
ADDV.D vA,vB,vC # Do add
SV vA, rA       # Store result
```
Gather/Scatter Operations

- Gather/scatter operations often implemented in hardware to handle sparse matrices
- Vector loads and stores use an index vector which is added to the base register to generate the addresses

<table>
<thead>
<tr>
<th>Index Vector</th>
<th>Data Vector</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.14</td>
<td>3.14</td>
</tr>
<tr>
<td>3</td>
<td>6.5</td>
<td>0.0</td>
</tr>
<tr>
<td>7</td>
<td>71.2</td>
<td>6.5</td>
</tr>
<tr>
<td>8</td>
<td>2.71</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>71.2</td>
</tr>
<tr>
<td></td>
<td>2.7</td>
<td>2.7</td>
</tr>
</tbody>
</table>
Conditional Operations in a Loop

What if some operations should not be executed on a vector (based on a dynamically-determined condition)?

```
loop: if (a[i] != 0) then b[i] = a[i] * b[i]
goto loop
```

Idea: Masked operations

- VMASK register is a bit mask determining which data element should not be acted upon
  - **VLD V0 = A**
  - **VLD V1 = B**
  - **VMASK = (V0 != 0)**
  - **VMUL V1 = V0 * V1**
  - **VST B = V1**

- Does this look familiar? This is essentially **predicated execution**.
Another Example with Masking

for (i = 0; i < 64; ++i)
    if (a[i] >= b[i]) then c[i] = a[i]
    else c[i] = b[i]

Steps to execute loop

1. Compare A, B to get VMASK
2. Masked store of A into C
3. Complement VMASK
4. Masked store of B into C

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>VMASK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>-5</td>
<td>-4</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-3</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>-7</td>
<td>-8</td>
<td>1</td>
</tr>
</tbody>
</table>
Masked Vector Instructions

Simple Implementation
- execute all N operations, turn off result writeback according to mask

Density-Time Implementation
- scan mask vector and only execute elements with non-zero masks

Slide credit: Krste Asanovic
Some Issues

- Stride and banking
  - As long as they are relatively prime to each other and there are enough banks to cover bank access latency, consecutive accesses proceed in parallel.

- Storage of a matrix
  - **Row major**: Consecutive elements in a row are laid out consecutively in memory.
  - **Column major**: Consecutive elements in a column are laid out consecutively in memory.
  - You need to change the stride when accessing a row versus column.
Matrix multiplication

A & B, both in row major order

\[
\begin{array}{cccccc}
    & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
A \rightarrow & 6 & 7 & 8 & 9 & 10 & 11 \\
\end{array}
\]

\[
\begin{array}{ccccccc}
    & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
B \rightarrow & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \\
\end{array}
\]

\[
A_{6x10} \times B_{10x4} \rightarrow C_{4x10} \quad \text{(dot products of rows & columns of A&B)}
\]

A:
- Load A into a vector register V1
  \rightarrow each time you need to increment the address by 1 to access the next column
  \rightarrow First matrix accesses have a \textit{stride of 1}

B:
- Load B into a vector-register V2
  \rightarrow each time you need to increment by 10
  \rightarrow \textit{stride of 10}

Different strides can lead to bank conflicts.
\rightarrow How do you minimize them?
Array vs. Vector Processors, Revisited

- Array vs. vector processor distinction is a “purist’s” distinction

- Most “modern” SIMD processors are a combination of both
  - They exploit data parallelism in both time and space
Remember: Array vs. Vector Processors

**Array Processor**
- **LD**: VR ← A[3:0]
- **ADD**: VR ← VR, 1
- **MUL**: VR ← VR, 2
- **ST**: A[3:0] ← VR

**Vector Processor**
- **LD**: A[3:0] ← VR
- **ADD**: VR ← VR, 1
- **MUL**: VR ← VR, 2
- **ST**: A[3:0] ← VR

**Instruction Stream**
- **Same op @ same time**
  - LD0, LD1, LD2, LD3
  - AD0, AD1, AD2, AD3
  - MU0, MU1, MU2, MU3
  - ST0, ST1, ST2, ST3

- **Different ops @ same space**
  - LD0, AD1, MU0, ST0
  - LD3, AD2, MU1, ST0

- **Different ops @ time**
  - LD0, AD0, MU0, ST0

- **Same op @ space**
  - LD0, AD1, MU0, ST0
  - LD3, AD2, MU1, ST0

**Space**
Vector Instruction Execution

ADDV C,A,B

Execution using one pipelined functional unit


Execution using four pipelined functional units


Slide credit: Krste Asanovic
Vector Unit Structure

- Vector Registers
  - Elements 0, 4, 8, ...
  - Elements 1, 5, 9, ...
  - Elements 2, 6, 10, ...
  - Elements 3, 7, 11, ...

- Memory Subsystem

- Lane

Functional Unit

Slide credit: Krste Asanovic
Vector Instruction Level Parallelism

Can overlap execution of multiple vector instructions

- example machine has 32 elements per vector register and 8 lanes
- Complete 24 operations/cycle while issuing 1 short instruction/cycle

Load Unit

Multiply Unit

Add Unit

Instruction issue

Slide credit: Krste Asanovic
Automatic Code Vectorization

Scalar Sequential Code

Vectorized Code

for (i=0; i < N; i++)
    C[i] = A[i] + B[i];

Vectorization is a compile-time reordering of operation sequencing ⇒ requires extensive loop dependence analysis
Vector/SIMD Processing Summary

- Vector/SIMD machines good at exploiting regular data-level parallelism
  - Same operation performed on many data elements
  - Improve performance, simplify design (no intra-vector dependencies)

- Performance improvement limited by vectorizability of code
  - Scalar operations limit vector machine performance
  - Amdahl’s Law
  - CRAY-1 was the fastest SCALAR machine at its time!

- Many existing ISAs include (vector-like) SIMD operations
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD
SIMD Operations in Modern ISAs
Intel Pentium MMX Operations

- Idea: One instruction operates on multiple data elements simultaneously
  - Alas array processing (yet much more limited)
  - Designed with multimedia (graphics) operations in mind

No VLEN register
Opcode determines data type:
- 8 8-bit bytes
- 4 16-bit words
- 2 32-bit doublewords
- 1 64-bit quadword

Stride always equal to 1.


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).
MMX Example: Image Overlaying (I)

Figure 8. Chroma keying: image overlay using a background color.

PCMPIEQB MM1, MM3

<table>
<thead>
<tr>
<th>MM1</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM3</td>
<td>X7!</td>
<td>X6!</td>
<td>X5!</td>
<td>X4!</td>
<td>X3!</td>
<td>X2!</td>
<td>X1</td>
<td>X0</td>
</tr>
<tr>
<td></td>
<td>blue</td>
<td>blue</td>
<td>blue</td>
<td>blue</td>
<td>blue</td>
<td>blue</td>
<td>blue</td>
<td>blue</td>
</tr>
<tr>
<td>MM1</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0xFFFF</td>
<td>0xFFFF</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0xFFFF</td>
<td>0xFFFF</td>
</tr>
</tbody>
</table>

Figure 9. Generating the selection bit mask.
MMX Example: Image Overlaying (II)

Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

```
Movq mm3, mem1  /* Load eight pixels from woman's image
Movq mm4, mem2  /* Load eight pixels from the blossom image
Pcmpaqb mm1, mm3
Pand mm4, mm1
Pandn mm1, mm3
Por mm4, mm1
```

Figure 11. MMX code sequence for performing a conditional select.