Computer Architecture:
SIMD and GPUs (Part II)

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

- These slides are partly from 18-447 Spring 2013, Computer Architecture, Lecture 19: SIMD and GPUs

- Video of the part related to only SIMD and GPUs:
  - [http://www.youtube.com/watch?v=dl5TZ4-oao0&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=19](http://www.youtube.com/watch?v=dl5TZ4-oao0&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=19)
Readings for Today


- See slides today for more readings (optional but recommended)
Today

- SIMD Processing
- GPU Fundamentals
- VLIW
Approaches to (Instruction-Level) Concurrency

- Pipelined execution
- Out-of-order execution
- Dataflow (at the ISA level)
- SIMD Processing
- VLIW

- Systolic Arrays
- Decoupled Access Execute
Review: 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
Review: SIMD Array Processing vs. VLIW

VLIW
Review: SIMD Array Processing vs. VLIW

- Array processor

![Diagram showing program counter and instruction execution]

Program Counter

```
add VR, VR, 1
```

VLEN = 4

Instruction Execution

- add VR[0], VR[0], 1
- add VR[1], VR[1], 1
- add VR[2], VR[2], 1
- add VR[3], VR[3], 1
Review: 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
Review: 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
Review: 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

Review: 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
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

![Diagram of vector registers]

- $V0,0$
- $V0,1$
- $V0,N-1$
- $V1,0$
- $V1,1$
- $V1,N-1$
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_

Slide credit: Krste Asanovic
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

0 1 2 3 4 5 6 7 8 9 A B C D E F

Slide credit: Krste Asanovic
Scalar Code Example

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

- Scalar code
  
  MOVIR0 = 50  1
  MOVA R1 = A  1
  MOVA R2 = B  1
  MOVA 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: $2 \times 11$ cycles
  - $4 + 50 \times 40 = 2004$ cycles

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

  - MOV VLEN = 50
  - MOV 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)

- 285 cycles
Vector Chaining

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

```
LV  v1
MULV v3, v1, v2
ADDV v5, v3, v4
```

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?**

VLD 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**
Gather/Scatter 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)

```c
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>
</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

|------|------|------|

- M[2] = 0
- M[1] = 1
- M[0] = 0

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

|------|------|------|

- M[2] = 0
- M[1] = 1
- M[0] = 0
- M[7] = 1

*Write data port*

---

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

\[
A_0 = \begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 \\
6 & 7 & 8 & 9 & 10 & 11
\end{bmatrix} \quad \quad \quad \quad \quad \\
B_0 = \begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \\
20 & 30 & 40 & 50 & & & & & & \\
& & & & & & & & & &
\end{bmatrix}
\]

\(A_{4 \times 6} \cdot B_{6 \times 10} \rightarrow C_{4 \times 10}\) (dot products of rows & columns of A & B)

A: Load \(A_0\) into a vector register \(V_1\)

\(\rightarrow\) each time you need to increment the address by 1 to access the next column

\(\rightarrow\) First matrix accesses have a stride of 1

B: Load \(B_0\) into a vector register \(V_2\)

\(\rightarrow\) each time you need to increment by 10

\(\rightarrow\) 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

**Instruction Stream**

- **LD** \( \rightarrow \) \( A[3:0] \)
- **ADD** \( \rightarrow \) \( VR, 1 \)
- **MUL** \( \rightarrow \) \( VR, 2 \)
- **ST** \( \rightarrow \) \( A[3:0] \)

**Time**

**Space**

**ARRAY PROCESSOR**

- **LD0**
- **AD0**
- **MU0**
- **ST0**

**VECTOR PROCESSOR**

- **LD0**
- **AD0**
- **MU0**
- **ST0**

**Same op @ same time**

**Different ops @ same space**

**Different ops @ time**

**Same op @ space**
Vector Instruction Execution

ADDV C, A, B

Execution using one pipelined functional unit


C[2]  
C[1]  
C[0]  

Execution using four pipelined functional units


C[8]  
C[4]  
C[0]  

C[9]  
C[5]  
C[1]  

C[10]  
C[6]  
C[2]  

C[11]  
C[7]  
C[3]  

Slide credit: Krste Asanovic
Vector Unit Structure

Functional Unit

Vector Registers

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

Lane

Memory Subsystem

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

Slide credit: Krste Asanovic
Vectorization is a compile-time reordering of operation sequencing. It 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
  - Ala 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.

MMX Example: Image Overlaying (I)

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

PCMPEQB 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!=blue</td>
<td>X6!=blue</td>
<td>X5=blue</td>
<td>X4=blue</td>
<td>X3!=blue</td>
<td>X2!=blue</td>
<td>X1=blue</td>
<td>X0=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
Pcmpeqb mm1, mm3
Pand mm4, mm1
Pandn mm1, mm3
Por mm4, mm1
```

Figure 11. MMX code sequence for performing a conditional select.
Graphics Processing Units
SIMD not Exposed to Programmer (SIMT)
High-Level View of a GPU
Concept of “Thread Warps” and SIMT

- **Warp**: A set of threads that execute the same instruction (on different data elements) $\rightarrow$ SIMT (Nvidia-speak)
- All threads run the same kernel
- **Warp**: The threads that run lengthwise in a woven fabric ...

![Diagram of Thread Warps and SIMD Pipeline]
Loop Iterations as Threads

Scalar Sequential Code

Vectorized Code

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

Slide credit: Krste Asanovic
SIMT Memory Access

- Same instruction in different threads uses thread id to index and access different data elements.

Let’s assume $N=16$, blockDim=4 $\rightarrow$ 4 blocks

Slide credit: Hyesoon Kim
Sample GPU SIMT Code (Simplified)

CPU code

```c
for (ii = 0; ii < 100; ++ii) {
}
```

CUDA code

```c
// there are 100 threads
__global__ void KernelFunction(...) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    int varA = aa[tid];
    int varB = bb[tid];
    C[tid] = varA + varB;
}
```

Slide credit: Hyesoon Kim
Sample GPU Program (Less Simplified)

### CPU Program

```c
void add_matrix
(float *a, float *b, float *c, int N) {
    int index;
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j) {
            index = i + j*N;
            c[index] = a[index] + b[index];
        }
}

int main() {
    add_matrix(a, b, c, N);
}
```

### GPU Program

```c
__global__ add_matrix
(float *a, float *b, float *c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if (i < N && j < N)
        c[index] = a[index] + b[index];
}

int main() {
    dim3 dimBlock( blocksize, blocksize);
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    add_matrix<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```
Latency Hiding with “Thread Warps”

- Warp: A set of threads that execute the same instruction (on different data elements)

- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No branch prediction)
  - Interleave warp execution to hide latencies

- Register values of all threads stay in register file

- No OS context switching

- Memory latency hiding
  - Graphics has millions of pixels
Warp-based SIMD vs. Traditional SIMD

- Traditional SIMD contains a single thread
  - Lock step
  - Programming model is SIMD (no threads) → SW needs to know vector length
  - ISA contains vector/SIMD instructions

- Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Does not have to be lock step
  - Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
    - SW does not need to know vector length
    - Enables memory and branch latency tolerance
  - ISA is scalar → vector instructions formed dynamically
  - Essentially, it is SPMD programming model implemented on SIMD hardware
SPMD

- Single procedure/program, multiple data
  - This is a programming model rather than computer organization

- Each processing element executes the same procedure, except on different data elements
  - Procedures can synchronize at certain points in program, e.g. barriers

- Essentially, multiple instruction streams execute the same program
  - Each program/procedure can 1) execute a different control-flow path, 2) work on different data, at run-time
  - Many scientific applications programmed this way and run on MIMD computers (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD computer
We did not cover the following slides in lecture. These are for your preparation for the next lecture.
Branch Divergence Problem in Warp-based SIMD

- SPMD Execution on SIMD Hardware
  - NVIDIA calls this “Single Instruction, Multiple Thread” ("SIMT") execution

Slide credit: Tor Aamodt
Control Flow Problem in GPUs/SIMD

- GPU uses SIMD pipeline to save area on control logic.
  - Group scalar threads into warps

- Branch divergence occurs when threads inside warps branch to different execution paths.

Slide credit: Tor Aamodt
Branch Divergence Handling (I)

Slide credit: Tor Aamodt
Branch Divergence Handling (II)

A;
if (some condition) {
    B;
} else {
    C;
}
D;

A; if (some condition) {
    B;
} else {
    C;
}
D;

Control Flow Stack

<table>
<thead>
<tr>
<th>Next PC</th>
<th>Recv PC</th>
<th>Amask</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>--</td>
<td>1111</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>1110</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>0001</td>
</tr>
</tbody>
</table>

Execution Sequence

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>B</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Time

TOS

Slide credit: Tor Aamodt
Dynamic Warp Formation

- **Idea:** Dynamically merge threads executing the same instruction (after branch divergence)
- **Form new warp at divergence**
  - Enough threads branching to each path to create full new warps
Dynamic Warp Formation/Merging

- **Idea:** Dynamically merge threads executing the same instruction (after branch divergence)

Dynamic Warp Formation Example

A new warp created from scalar threads of both Warp x and y executing at Basic Block D

Baseline

Dynamic Warp Formation

Legend

Execution of Warp x at Basic Block A

Execution of Warp y at Basic Block A

A new warp created from scalar threads of both Warp x and y executing at Basic Block D

Slide credit: Tor Aamodt
What About Memory Divergence?

- Modern GPUs have caches
- Ideally: Want all threads in the warp to hit (without conflicting with each other)
- Problem: One thread in a warp can stall the entire warp if it misses in the cache.

- Need techniques to
  - Tolerate memory divergence
  - Integrate solutions to branch and memory divergence
NVIDIA GeForce GTX 285

- NVIDIA-speak:
  - 240 stream processors
  - “SIMT execution”

- Generic speak:
  - 30 cores
  - 8 SIMD functional units per core
NVIDIA GeForce GTX 285 “core”

- 64 KB of storage for fragment contexts (registers)
- SIMD functional unit, control shared across 8 units
- multiply-add
- multiply
- instruction stream decode
- execution context storage

Slide credit: Kayvon Fatahalian
NVIDIA GeForce GTX 285 “core”

- Groups of 32 threads share instruction stream (each group is a Warp)
- Up to 32 warps are simultaneously interleaved
- Up to 1024 thread contexts can be stored

Slide credit: Kayvon Fatahalian
There are 30 of these things on the GTX 285: 30,720 threads