Computer Architecture
Lecture 8: SIMD Processors and GPUs

Prof. Onur Mutlu
ETH Zürich
Fall 2017
18 October 2017
Agenda for Today & Next Few Lectures

- SIMD Processors
- GPUs
- Introduction to GPU Programming

Digitaltechnik (Spring 2017) YouTube videos
Lecture 19: Beginning of SIMD
https://youtu.be/XE9ogMPEMLw?t=1h11m42s
Lecture 20: SIMD Processors
https://youtu.be/hRHs7xIP0Sg?t=6m48s
Lecture 21: GPUs
https://youtu.be/MUPTdxl3JKs?t=3m03s
SIMD 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" (more appropriately, operations) are 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 using different spaces
  - **Vector processor**: Instruction operates on multiple data elements in consecutive time steps using the same space
Array vs. Vector Processors

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

**ARRAY PROCESSOR**
- **PE0**
- **PE1**
- **PE2**
- **PE3**

**VECTOR PROCESSOR**
- **LD**
- **ADD**
- **MUL**
- **ST**

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

- **Different ops @ time**
  - **LD0**
  - **AD0**
  - **LD1**
  - **AD1**
  - **MU0**
  - **LD2**
  - **AD2**
  - **MU1**
  - **LD3**
  - **AD3**
  - **MU2**
  - **ST0**
  - **AD3**
  - **MU3**
  - **ST1**
  - **MU3**
  - **ST2**
  - **ST3**

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

**Space**
SIMD Array Processing vs. VLIW

- VLIW: Multiple independent operations packed together by the compiler
SIMD Array Processing vs. VLIW

- Array processor: Single operation on multiple (different) data elements

![Diagram showing SIMD array processing and VLIW instructions]

- 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

VLEN = 4
Vector Processors

- A vector is a one-dimensional array of numbers.
- Many scientific/commercial programs use vectors:
  ```c
  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
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 registers/cache/memory
Vector Processor Advantages

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

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

+ Highly regular memory access pattern

+ 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
Vector Processing in More Depth
Vector Registers

- Each **vector data register** holds N M-bit values
- **Vector control registers**: VLEN, VSTR, VMASK
- Maximum VLEN can be N
  - Maximum number of elements stored in a vector register
- **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)

![Diagram of vector registers](image)

V0,0
V0,1
V0,N-1

V1,0
V1,1
V1,N-1
Vector Functional Units

- Use deep pipeline to execute element operations → fast clock cycle

- Control of deep pipeline is simple because elements in vector are independent

\[
V_1 \times V_2 \rightarrow V_3
\]

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
Loading/Storing Vectors from/to Memory

- Requires loading/storing multiple elements

- Elements separated from each other by a constant distance (stride)
  - Assume stride = 1 for now

- Elements can be loaded in consecutive cycles if we can start the load of one element per cycle
  - Can sustain a throughput of one element per cycle

- Question: How do we achieve this with a memory that takes more than 1 cycle to access?
- Answer: Bank the memory; interleave the elements across banks
Memory Banking

- Memory is divided into **banks** that can be accessed independently; banks share address and data buses (to minimize pin cost)
- Can start and complete one bank access per cycle
- **Can sustain N parallel accesses if all N go to different banks**

![Diagram of memory banking](Picture credit: Derek Chiou)
Vector Memory System

- Next address = Previous address + Stride
- If stride = 1 & consecutive elements interleaved across banks & number of banks >= bank latency, then can sustain 1 element/cycle throughput
Scalar Code Example

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

- Scalar code (instruction and its latency)
  
  MOVI R0 = 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 (In Order)

- 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: consecutive words are stored in consecutive banks)
  - 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 (each instruction and its latency):
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVI VLEN = 50</td>
<td>1</td>
</tr>
<tr>
<td>MOV VSTR = 1</td>
<td>1</td>
</tr>
<tr>
<td>VLD V0 = A</td>
<td>11 + VLEN - 1</td>
</tr>
<tr>
<td>VLD V1 = B</td>
<td>11 + VLEN - 1</td>
</tr>
<tr>
<td>VADD V2 = V0 + V1</td>
<td>4 + VLEN - 1</td>
</tr>
<tr>
<td>VSHFR V3 = V2 &gt;&gt; 1</td>
<td>1 + VLEN - 1</td>
</tr>
<tr>
<td>VST C = V3</td>
<td>11 + VLEN - 1</td>
</tr>
</tbody>
</table>

7 dynamic instructions
Basic Vector Code Performance

- Assume no chaining (no vector data forwarding)
  - i.e., output of a vector functional unit cannot be used as the direct input of another
  - The entire vector register needs to be ready before any element of it can be used as part of another operation
- 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

Slide credit: Krste Asanovic
- **Vector chaining**: Data forwarding from one vector functional unit to another

  These two VLDs cannot be pipelined. WHY?

  VLD and VST cannot be pipelined. WHY?

- **182 cycles**

  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
- 19X perf. improvement!
Questions (I)

- What if # data elements > # elements in a vector register?
  - Idea: **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)
  - Idea: **Use indirection to combine/pack elements into vector registers**
  - Called **scatter/gather operations**
Gather/Scatter Operations

Want to vectorize loops with indirect accesses:

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

Indexed load instruction (Gather)

```plaintext
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 vectors (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 (to Store)</th>
<th>Stored Vector (in Memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3.14</td>
<td>Base+0 3.14</td>
</tr>
<tr>
<td>2</td>
<td>6.5</td>
<td>Base+1 X</td>
</tr>
<tr>
<td>6</td>
<td>71.2</td>
<td>Base+2 6.5</td>
</tr>
<tr>
<td>7</td>
<td>2.71</td>
<td>Base+3 X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base+4 X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base+5 X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base+6 71.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base+7 2.71</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)?

```c
loop: for (i=0; i<N; i++)
    if (a[i] != 0) then b[i]=a[i]*b[i]
```

- 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
  - This is *predicated execution*. Execution is *predicated* on mask bit.
Another Example with Masking

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

Steps to execute the loop in SIMD code

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

\[
\begin{align*}
M[0] &= 0 & & C[0]
\end{align*}
\]

Write Enable

Write data port

Density-Time Implementation

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

\[
\begin{align*}
M[7] &= 1 \\
M[6] &= 0 \\
M[5] &= 1 \\
M[4] &= 1 \\
M[3] &= 0 \\
M[2] &= 0 \\
M[1] &= 1 \\
M[0] &= 0 \\
C[1] & \rightarrow & Write \ data \ port
\end{align*}
\]

Which one is better?

Tradeoffs?

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, we can sustain 1 element/cycle throughput

- 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

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

\[A \times B_{8x10} \rightarrow C_{4x10}\] (dot products of rows & columns of A & B)

A: Load A into a vector register \( V_1 \)
- each time you need to increment the address by 1 to access the next column
- First matrix accesses have a stride of 1

B: Load B into a vector register \( V_2 \)
- each time you need to increment by 10
- Stride of 10

Different strides can lead to bank conflicts.
- How do you minimize them?
Minimizing Bank Conflicts

- More banks

- Better data layout to match the access pattern
  - Is this always possible?

- Better mapping of address to bank
  - E.g., randomized mapping
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
  - GPUs are a prime example we will cover in a bit more detail
Remember: Array vs. Vector Processors

### Instruction Stream

<table>
<thead>
<tr>
<th>LD</th>
<th>VR ← A[3:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>VR ← VR, 1</td>
</tr>
<tr>
<td>MUL</td>
<td>VR ← VR, 2</td>
</tr>
<tr>
<td>ST</td>
<td>A[3:0] ← VR</td>
</tr>
</tbody>
</table>

#### Time

**ARRAY PROCESSOR**

- LD0
- AD0
- MU0
- ST0

- LD1
- AD1
- MU1
- ST1

- LD2
- AD2
- MU2
- ST2

- LD3
- AD3
- MU3
- ST3

**VECTOR PROCESSOR**

- LD
- ADD
- MUL
- ST

### Space

- **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
  - AD0
  - LD1
  - AD1

- **Different ops @ time**
  - LD2
  - AD2
  - LD3
  - AD3

- **Same op @ space**
  - MU0
  - MU1
  - MU2
  - MU3

- **ST0**
  - ST1
  - ST2
  - ST3
Vector Instruction Execution

VADD A,B \rightarrow C

Execution using one pipelined functional unit

\begin{align*}
\end{align*}

\begin{align*}
C[0] & \rightarrow C[0] \\
\end{align*}

Execution using four pipelined functional units

\begin{align*}
\end{align*}

\begin{align*}
\end{align*}

\begin{align*}
A[22] & \rightarrow B[22] \\
\end{align*}

\begin{align*}
A[27] & \rightarrow B[27] \\
\end{align*}

\begin{align*}
C[0] & \rightarrow C[0] \\
\end{align*}

\begin{align*}
C[10] & \rightarrow C[1] \\
\end{align*}

\begin{align*}
\end{align*}

Slide credit: Krste Asanovic
Vector Unit Structure

Partitioned Vector Registers

Elements 0, 4, 8, ...

Elements 1, 5, 9, ...

Elements 2, 6, 10, ...

Elements 3, 7, 11, ...

Functional Unit

Memory Subsystem

Lane

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
- Completes 24 operations/cycle while issuing 1 vector instruction/cycle

Load Unit

Multiplying Unit

Adding Unit

Time

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

Slide credit: Krste Asanovic
Vector/SIMD Processing Summary

- Vector/SIMD machines are 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
  - Remember 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
SIMD ISA Extensions

- Single Instruction Multiple Data (SIMD) extension instructions
  - Single instruction acts on multiple pieces of data at once
  - Common application: graphics
  - Perform short arithmetic operations (also called packed arithmetic)

- For example: add four 8-bit numbers
- Must modify ALU to eliminate carries between 8-bit values

\[
\text{\texttt{padd8 $s2, $s0, $s1}}
\]

<table>
<thead>
<tr>
<th></th>
<th>32</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
<th>Bit position</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(a_3)</td>
<td>(a_2)</td>
<td>(a_1)</td>
<td>(a_0)</td>
<td>(s0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td>(b_3)</td>
<td>(b_2)</td>
<td>(b_1)</td>
<td>(b_0)</td>
<td>(s1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) + (b)</td>
<td>(a_3 + b_3)</td>
<td>(a_2 + b_2)</td>
<td>(a_1 + b_1)</td>
<td>(a_0 + b_0)</td>
<td>(s2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 is always equal to 1.

Goal: Overlay the human in image 1 on top of the background in image 2.

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

```
<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>
</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>
</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>
</tr>
</tbody>
</table>
```

Figure 9. Generating the selection bit mask.

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
Pcmpreqb mm1, mm3
Pand mm4, mm1
Pandn mm1, mm3
Por mm4, mm1
```

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

GPUs (Graphics Processing Units)
GPUs are SIMD Engines Underneath

- The instruction pipeline operates like a SIMD pipeline (e.g., an array processor)
- However, the programming is done using threads, NOT SIMD instructions
- To understand this, let’s go back to our parallelizable code example

- But, before that, let’s distinguish between
  - Programming Model (Software)
    - vs.
  - Execution Model (Hardware)
Programming Model vs. Hardware Execution Model

- Programming Model refers to **how the programmer expresses the code**
  - E.g., Sequential (von Neumann), Data Parallel (SIMD), Dataflow, Multi-threaded (MIMD, SPMD), ...

- Execution Model refers to **how the hardware executes the code underneath**
  - E.g., Out-of-order execution, Vector processor, Array processor, Dataflow processor, Multiprocessor, Multithreaded processor, ...

- Execution Model can be very different from the Programming Model
  - E.g., von Neumann model implemented by an OoO processor
  - E.g., SPMD model implemented by a SIMD processor (a GPU)
How Can You Exploit Parallelism Here?

Let’s examine three programming options to exploit instruction-level parallelism present in this sequential code:

1. Sequential (SISD)
2. Data-Parallel (SIMD)
3. Multithreaded (MIMD/SPMD)

```c
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```
Prog. Model 1: Sequential (SISD)

Scalar Sequential Code

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

- Can be executed on a:
  - Pipelined processor
  - Out-of-order execution processor
    - Independent instructions executed when ready
    - Different iterations are present in the instruction window and can execute in parallel in multiple functional units
    - In other words, the loop is dynamically unrolled by the hardware
  - Superscalar or VLIW processor
    - Can fetch and execute multiple instructions per cycle
Scalar Sequential Code

Vector Instruction

Vectorized Code

Iter. 1

load

load

add

store

Iter. 2

load

load

add

store

load

load

add

store

load

load

add

store

Iter. 3

load

load

add

store

load

load

add

store

load

load

add

store

Iter. 4

for (i=0; i < N; i++)

C[i] = A[i] + B[i];

Vector Instruction

VLD

A \rightarrow V1

VLD

B \rightarrow V2

VADD

V1 + V2 \rightarrow V3

VST

V3 \rightarrow C

Realization: Each iteration is independent

Idea: Programmer or compiler generates a SIMD instruction to execute the same instruction from all iterations across different data

Best executed by a SIMD processor (vector, array)
Prog. Model 3: Multithreaded

Scalar Sequential Code

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

**Realization:** Each iteration is independent

**Idea:** Programmer or compiler generates a thread to execute each iteration. Each thread does the same thing (but on different data)

Can be executed on a MIMD machine
Prog. Model 3: Multithreaded

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

Realization: Each iteration is independent

This particular model is also called:

SPMD: Single Program Multiple Data

Can be executed on a SIMT machine
Single Instruction Multiple Thread
A GPU is a SIMD (SIMT) Machine

- Except it is not programmed using SIMD instructions

- It is programmed using threads (SPMD programming model)
  - Each thread executes the same code but operates a different piece of data
  - Each thread has its own context (i.e., can be treated/restarted/executed independently)

- A set of threads executing the same instruction are dynamically grouped into a warp (wavefront) by the hardware
  - A warp is essentially a SIMD operation formed by hardware!
for (i=0; i < N; i++)
C[i] = A[i] + B[i];

This particular model is also called:

**SPMD: Single Program Multiple Data**

A GPU executes it using the SIMT model:
Single Instruction Multiple Thread
Graphics Processing Units
SIMD not Exposed to Programmer (SIMT)
SIMD vs. SIMT Execution Model

- **SIMD**: A single *sequential instruction stream* of *SIMD instructions* → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN

- **SIMT**: *Multiple instruction streams* of *scalar instructions* → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

- **Two Major SIMT Advantages**:
  - *Can treat each thread separately* → i.e., can execute each thread independently (on any type of scalar pipeline) → *MIMD processing*
  - *Can group threads into warps flexibly* → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing
Multithreading of Warps

- Assume a warp consists of 32 threads
- If you have 32K iterations, and 1 iteration/thread → 1K warps
- Warps can be interleaved on the same pipeline → Fine grained multithreading of warps

\[
\text{for } (i=0; i < N; i++) \\
C[i] = A[i] + B[i];
\]
Warps and Warp-Level FGMT

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

![Diagram of thread warps and SIMD pipeline]
High-Level View of a GPU
Latency Hiding via Warp-Level FGMT

- 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 interlocking)
  - Interleave warp execution to hide latencies

- Register values of all threads stay in register file

- FGMT enables long latency tolerance
  - Millions of pixels
Warp Execution (Recall the Slide)

32-thread warp executing ADD $A[tid], B[tid] \rightarrow C[tid]$

Execution using one pipelined functional unit

Execution using four pipelined functional units


SIMD Execution Unit Structure

- **Functional Unit**
- **Memory Subsystem**
- **Lane**
- **Registers for each Thread**
  - Registers for thread IDs 0, 4, 8, ...
  - Registers for thread IDs 1, 5, 9, ...
  - Registers for thread IDs 2, 6, 10, ...
  - Registers for thread IDs 3, 7, 11, ...

Slide credit: Krste Asanovic
Warp Instruction Level Parallelism

Can overlap execution of multiple instructions

- Example machine has 32 threads per warp and 8 lanes
- Completes 24 operations/cycle while issuing 1 warp/cycle

Slide credit: Krste Asanovic
Same instruction in different threads uses thread id to index and access different data elements.

Let’s assume $N=16$, 4 threads per warp $\rightarrow$ 4 warps.

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

CPU code

```c
for (ii = 0; ii < 100000; ++ii) {
}
```

CUDA code

```c
__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( blockDim, blockDim);
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N);
}
```

Slide credit: Hyesoon Kim
Warp-based SIMD vs. Traditional SIMD

- Traditional SIMD contains a single thread
  - Sequential instruction execution; lock-step operations in a SIMD instruction
  - Programming model is SIMD (no extra 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 multithreading and flexible dynamic grouping of threads
  - ISA is scalar → SIMD operations can be 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 1) works on different data, 2) can execute a different control-flow path, at run-time
  - Many scientific applications are programmed this way and run on MIMD hardware (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD hardware
SIMD vs. SIMT Execution Model

- **SIMD**: A single **sequential instruction stream** of SIMD instructions → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN

- **SIMT**: **Multiple instruction streams** of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

- **Two Major SIMT Advantages:**
  - Can treat each thread separately → i.e., can execute each thread independently on any type of scalar pipeline → MIMD processing
  - Can group threads into warps flexibly → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing
Threads Can Take Different Paths in Warp-based SIMD

- Each thread can have conditional control flow instructions
- Threads can execute different control flow paths
Control Flow Problem in GPUs/SIMT

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

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

This is the same as conditional/predicated/masked execution. Recall the Vector Mask and Masked Vector Operations?

Slide credit: Tor Aamodt
Remember: Each Thread Is Independent

- Two Major SIMT Advantages:
  - Can treat each thread separately → i.e., can execute each thread independently on any type of scalar pipeline → MIMD processing
  - Can group threads into warps flexibly → i.e., can group threads that are supposed to truly execute the same instruction → dynamically obtain and maximize benefits of SIMD processing

- If we have many threads
- We can find individual threads that are at the same PC
- And, group them together into a single warp dynamically
- This reduces “divergence” → improves SIMD utilization
  - SIMD utilization: fraction of SIMD lanes executing a useful operation (i.e., executing an active thread)
Dynamic Warp Formation/Merging

- **Idea:** Dynamically merge threads executing the same instruction (after branch divergence)
- **Form new warps from warps that are waiting**
  - Enough threads branching to each path enables the creation of full new warps
Dynamic Warp Formation/Merging

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

Dynamic Warp Formation Example

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
Hardware Constraints Limit Flexibility of Warp Grouping

Can you move any thread flexibly to any lane?
An Example GPU
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”

- = SIMD functional unit, control shared across 8 units
  - = multiply-add
  - = multiply
- = instruction stream decode
- = execution context storage

64 KB of storage for thread contexts (registers)

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

64 KB of storage for thread contexts (registers)

Slide credit: Kayvon Fatahalian
30 cores on the GTX 285: 30,720 threads

Slide credit: Kayvon Fatahalian