P&S Heterogeneous Systems

SIMD Processing and GPUs

Dr. Juan Gómez Luna
Prof. Onur Mutlu

ETH Zürich
Spring 2022
22 March 2022
The end of Moore’s law created the need for heterogeneous systems

- More suitable devices for each type of workload
- Increased performance and energy efficiency

Recall: 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 operation 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 operation-level parallelism on different data
  - Same operation concurrently applied to different pieces of data
  - A form of ILP where instruction happens to be the same across data
SIMD Processing

- Single instruction operates on multiple data elements
  - In time or in space
- Multiple processing elements (PEs), i.e., execution units

- Time-space duality

  - **Array processor**: Instruction operates on multiple data elements at the same time using different spaces (PEs)

  - **Vector processor**: Instruction operates on multiple data elements in consecutive time steps using the same space (PE)
Array vs. Vector Processors

Instruction Stream

<table>
<thead>
<tr>
<th>LD</th>
<th>ADD</th>
<th>MUL</th>
<th>ST</th>
</tr>
</thead>
</table>

Time

Space

ARRAY PROCESSOR

LD0 | LD1 | LD2 | LD3
AD0 | AD1 | AD2 | AD3
MU0 | MU1 | MU2 | MU3
ST0 | ST1 | ST2 | ST3

VECTOR PROCESSOR

LD0 | ADD | MUL | ST
LD1 | AD0 | MU0
LD2 | AD1 | MU1 | ST0
LD3 | AD2 | MU2 | ST1
AD3 | MU3 | ST2 | ST3

- Same op @ same time
- Different ops @ same space
- Different ops @ time
- Same op @ space
Vector Processors (I)

- A vector is a one-dimensional array of numbers
- Many scientific/commercial programs use vectors

```plaintext
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 in memory between two elements of a vector
Vector Stride Example: Matrix Multiply

- A and B matrices, both stored in memory in row-major order

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

\[A_{4\times6} \ B_{6\times10} \rightarrow C_{4\times10}\]

Dot product of each row vector of A with each column vector of B

- Load A’s row 0 (A_{00} through A_{05}) into vector register \( V_1 \)
  - Each time, increment address by 1 to access the next column
  - Accesses have a stride of 1

- Load B’s column 0 (B_{00} through B_{50}) into vector register \( V_2 \)
  - Each time, increment address by 10 to access the next row
  - Accesses have a stride of 10
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 needed within a vector
  - **No control flow within a vector**
  - **Known stride allows easy address calculation** for all vector elements
    - Enables **easy loading** (or even early loading, i.e., prefetching) of vectors into registers/cache/memory
Recall: 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 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)
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 concurrent accesses if all N go to different banks**.

![Diagram of Memory Banking](Picture credit: Derek Chiou)
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>MOV1 VLEN = 50</td>
<td>1</td>
</tr>
<tr>
<td>MOV1 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 Chainning: 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
  - 19X perf. improvement!
Conditional Operations in a Loop

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

```
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>
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 format 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
Vector Stride Example: Matrix Multiply

- A and B matrices, both stored in memory in **row-major order**

```
<table>
<thead>
<tr>
<th>A_0</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>
```

\( A_{4 \times 6} \rightarrow C_{4 \times 10} \)

Dot product of each row vector of A with each column vector of B

- Load A’s row 0 (A_{00} through A_{05}) into vector register V_1
  - Each time, increment address by 1 to access the next column
  - Accesses have a **stride of 1**

- Load B’s column 0 (B_{00} through B_{50}) into vector register V_2
  - Each time, increment address by 10 to access the next row
  - Accesses have a **stride of 10**

**Different strides can lead to bank conflicts**

**How do we minimize them?**

![Linear Memory Diagram](image-url)
Recall: 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 concurrent accesses if all N go to different banks
Minimizing Bank Conflicts

- More banks
- More ports in each bank
- **Better data layout** to match the access pattern
  - Is this always possible?
- **Better mapping of address to bank**
  - E.g., randomized mapping
Minimizing Bank Conflicts: Recommended Reading

PSEUDO-RANDOMLY INTERLEAVED MEMORY

B. Ramakrishna Rau
Hewlett Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303

ABSTRACT

Interleaved memories are often used to provide the high bandwidth needed by multiprocessors and high performance uniprocessors such as vector and VLIW processors. The manner in which memory locations are distributed across the memory modules has a significant influence on whether, and for which types of reference patterns, the full bandwidth of the memory system is achieved. The most common interleaved memory architecture is the sequentially interleaved memory in which successive memory locations are assigned to successive memory modules. Although such an architecture is the simplest to implement and provides good performance with strides that are odd integers, it can degrade badly in the face of even strides, especially strides that are a power of two.

In a pseudo-randomly interleaved memory architecture, memory locations are assigned to the memory modules in some pseudo-random fashion in the hope that those sequences of references, which are likely to occur in practice, will end up being evenly distributed across the memory modules. The notion of polynomial interleaving modulo an irreducible polynomial is introduced as a way of achieving pseudo-random interleaving with certain attractive and provable properties. The theory behind this scheme is developed and the results of simulations are presented.

Keywords: supercomputer memory, parallel memory, interleaved memory, hashed memory, pseudo-random interleaving, memory buffering.

The conventional solution is to provide each processor with a data cache constructed out of SRAM. The problem is maintaining cache coherency, at high request rates, across multiple private caches in a multiprocessor system. The alternative is to use a shared cache if the additional delay incurred in going through the processor-cache interconnect is acceptable. The problem here is that the bandwidth, even with SRAM chips, is inadequate unless some form of interleaving is employed in the cache. So once again, the interleaving scheme used is an issue. Furthermore, data caches are susceptible to problems arising out of the lack of spatial and/or data locality in the data reference pattern of many applications. This phenomenon has been studied and reported elsewhere, e.g., in [4,5]. Since data caches are essential to achieving good performance on scalar computations with little parallelism, the right compromise is to provide a data cache that can be bypassed when referencing data structures with poor locality. This is the solution employed in various recent products such as the Convex C-1 and Intel's i860.

Interleaved memory systems. Whether or not a data cache is present, it is important to provide a memory system with bandwidth to match the processors. This is done by organizing the memory system as multiple memory modules which can operate in parallel. The manner in which memory locations are distributed across the memory modules has a significant influence on whether, and for which types of reference patterns, the full bandwidth of the memory system is achieved.

Engineering and scientific applications include

SIMD Operations in Modern ISAs
MMX Example: Image Overlaying (I)

- Goal: Overlay the human in image x on top of the background in image y

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

```plaintext
for (i=0; i<image_size; i++) { 
  if (x[i] == Blue) new_image[i] = y[i];
  else new_image[i] = x[i];
}
```

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.

```c
for (i=0; i<image_size; i++) {
  if (x[i] == Blue) new_image[i] = y[i];
  else new_image[i] = x[i];
}
```

Figure 11. MMX code sequence for performing 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
```
From MMX to AMX in x86 ISA

- **MMX**
  - 64-bit MMX registers for integers

- **SSE (Streaming SIMD Extensions)**
  - SSE-1: 128-bit XMM registers for integers and single-precision floating point
  - SSE-2: Double-precision floating point
  - SSE-3, SSSE-3 (supplemental): New instructions
  - SSE-4: New instructions (not multimedia specific), shuffle operations

- **AVX (Advanced Vector Extensions)**
  - AVX: 256-bit floating point
  - AVX2: 256-bit floating point with **FMA (Fused Multiply Add)**
  - AVX-512: 512-bit

- **AMX (Advanced Matrix Extensions)**
  - Designed for AI/ML workloads
  - 2-dimensional registers
  - Tiled matrix multiply unit (TMUL)

Lecture on SIMD Processing

CRAY X-MP-28 @ ETH (CAB, E Floor)

Digital Design & Comp. Arch. - Lecture 20: SIMD Processing (Vector and Array Processors) (Spring’21)

Onur Mutlu Lectures
19.2K subscribers

https://youtu.be/fP4kZ2Zx_84
SIMD Operations in Modern (Machine Learning) Accelerators
Cerebras’s Wafer Scale Engine (2019)

- The largest ML accelerator chip (2019)
- 400,000 cores

Cerebras WSE
1.2 Trillion transistors
46,225 mm²

Largest GPU
21.1 Billion transistors
815 mm²
NVIDIA TITAN V

https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning
https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning/
Cerebras’s Wafer Scale Engine-2 (2021)

- The largest ML accelerator chip (2021)
- 850,000 cores

**Cerebras WSE-2**
- 2.6 Trillion transistors
- 46,225 mm²

**Largest GPU**
- 54.2 Billion transistors
- 826 mm²
  - NVIDIA Ampere GA100

https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning

https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning/
Size, Place, and Route in Cerebras’s WSE

- Neural network mapping onto the whole wafer is a challenge

Multiple possible mappings

Different dies of the wafer work on different layers of the neural network: **MIMD** machine

An example mapping

Layers mapped on Wafer Scale Engine

James et al., “ISPD 2020 Physical Mapping of Neural Networks on a Wafer-Scale Deep Learning Accelerator.”
Recall: 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
A MIMD Machine with SIMD Processors (I)

- **MIMD** machine
  - Distributed memory (no shared memory)
  - 2D-mesh interconnection fabric

---

**SIMD processors**
- 4-way SIMD for 16-bit floating point operands
- 48 KB of local SRAM

**Figure 2. CS-1 Wafer Scale Engine (WSE).** A single wafer (rightmost) contains one CS-1 processor. Each processor is a collection of dies arranged in a 2D fashion (middle). Dies are then further subdivided into a grid of tiles. One die hosts thousands of computational cores, memory and routers (leftmost). There is no logical discontinuity between adjacent dies and there is no additional bandwidth penalty for crossing the die-die barrier. In total, there are 1.2 trillion transistors in an area of $462.25 \text{ cm}^2$.

Tensors, making use of tensor address generation hardware to efficiently access tensor data in memory. These play the role of nested loops and eliminate any loop overhead. There are enough memory banks to provide the bandwidth needed to fetch eight 16-bit words from memory and store four such words per cycle, enough to support SIMD-4, AXPY operations $y = y + a \times x$, where the operand $a$ is a scalar held in a register and $x$ and $y$ are tensors that stream to and from memory. Such an operation can be launched with a single instruction. The tensor operands can have more than four elements, so the instruction executes for multiple cycles.

In mixed precision with multiplications in fp16 and additions performed in fp32, the throughput is two FMACs per core per cycle. Purely 32-bit floating point computations run one FMAC per core per cycle. The theoretical peak performance of the system varies depending on the number of cores configured on the wafer, clock rate and power settings.

The core connects to a local router that has five bidirectional links, one to each of its four nearest neighbors and one to its own core. The router can move data into and out of these five links, in parallel, on every cycle. Even with scalar granularity, communication is efficient. The router has hardware queues for its connection to the core and for each of a set of virtual channels, avoiding deadlock. Communication between potentially distant processors occurs along predetermined routes. Routing is configured offline, as part of compilation; data travel along virtual channels that can be programatically reconfigured at run time. The fanout of data to multiple destinations is done through the routing; the router can forward an input word to any subset of its five output ports. There is no runtime software involved with communication. Arriving data are deposited by the hardware directly into memory or registers or routed to functional units as specified by the program.

An instruction with tensor operands can run synchronously or, at the discretion of the programmer, as a background thread that shares the datapath with other threads including the main one. A background thread runs a single tensor operation, as a single asynchronously running instruction. There is no context switch overhead. The registers and memory used by an asynchronous thread are those assigned by the programmer or compiler in the instruction, and these may not be overwritten until the thread terminates. Subsequent computation can be delayed until the thread terminates. The core supports nine concurrent threads of execution.

A stream of data to or from the fabric may be used as an input to a tensor operation, or as the destination for one. The hardware directly implements scheduling activities that would normally be performed by an operating system. This allows compact and efficient software implementations. For example, one core can be sending data from its local memory to another core; simultaneously it can receive data from another core while adding it to values stored in its local memory. All of this is accomplished using only two machine instructions that run as independent threads.

Code consists of tasks that react to events. Tasks are triggered by other tasks, or by arriving data words. The channel of the arriving word determines the code that is triggered. There is little delay between the completion of a task and the start of a subsequent task, as this is handled in hardware. Together with the SIMD operations 4-way SIMD fused-multiply-accumulate (FMAC) units.

**AXPY:**

\[
y = a \times x + y
\]
SAFARI Live Seminar: Sean Lie, 28 Feb 2022

Posted on January 19, 2022 by ewent

Join us for our SAFARI Live Seminar with Sean Lie, Cerebras Systems
Monday, February 28 2022 at 6:00 pm Zurich time (CET)

Sean Lie, co-founder and Chief Hardware Architect at Cerebras Systems
Thinking Outside the Die: Architecting the ML Accelerator of the Future

Livestream on YouTube Link

Heterogeneous Computing Systems

- The end of Moore’s law created the need for heterogeneous systems
  - More suitable devices for each type of workload
  - Increased performance and energy efficiency

GPUs (Graphics Processing Units)
108 cores on the A100
(Up to 128 cores in the full-blown chip)

40MB L2 cache

https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/
Recall: Array vs. Vector Processors

**Instruction Stream**

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

Vector Processor:
- LD0
- AD0
- MU0
- ST0
- LD1
- AD1
- MU1
- ST1
- LD2
- AD2
- MU2
- ST2
- LD3
- AD3
- MU3
- ST3

**Time**

Array:
- LD0
- LD1
- LD2
- LD3
- AD0
- AD1
- AD2
- AD3
- MU0
- MU1
- MU2
- MU3
- ST0
- ST1
- ST2
- ST3

Vector:
- LD0
- AD0
- MU0
- AD2
- MU1
- ST0
- AD3
- MU2
- ST1
- MU3
- ST2
- ST3

**Space**

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

Vector:
- Different ops @ time
- Same op @ space
NVIDIA A100 Core

- **19.5 TFLOPS Single Precision**
- **9.7 TFLOPS Double Precision**
- **312 TFLOPS for Deep Learning (Tensor cores)**

---

**SM**

<table>
<thead>
<tr>
<th>L1 Instruction Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp Scheduler (32 thread/clock)</td>
</tr>
<tr>
<td>Dispatch Unit (32 thread/clock)</td>
</tr>
</tbody>
</table>

**Register File (16,384 x 32-bit)**

**L0 Instruction Cache**

- Warp Scheduler (32 thread/clock)
- Dispatch Unit (32 thread/clock)

**192KB L1 Data / Shared Memory**

- Tex
- Tex
- Tex
- Tex

---

Recall: 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 concurrent accesses if all N go to different banks

---

**Diagram:**

- Banks are numbered from 0 to 15.
- MDR (Memory Data Register) and MAR (Memory Address Register) are connected to each bank.
- Data bus and Address bus connect to all banks.
- CPU interacts with the memory through the data and address buses.
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)
  - 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?

Scalar Sequential Code

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

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)
Prog. Model 1: Sequential (SISD)

Scalar Sequential Code

- 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
**Prog. Model 2: Data Parallel (SIMD)**

For \( i = 0; i < N; i++ \)

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

---

**Scalar Sequential Code**

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

**Vector Instruction**

```
load
load
add
store
```

**Vectorized Code**

```
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**

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

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

Scalar Sequential Code

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

Iter. 1
Iter. 2

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
Fine-Grained 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

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

![Diagram showing load, add, store operations for warps at PC X and PC X+2 with iteration indices iter. 19*32 + 1 and iter. 20*32 + 2]
Fine-Grained Multithreading
Fine-Grained Multithreading

- Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
  - Branch/instruction resolution latency overlapped with execution of other threads’ instructions

+ No logic needed for handling control and data dependences within a thread
  -- Single thread performance suffers
  -- Extra logic for keeping thread contexts
  -- Does not overlap latency if not enough threads to cover the whole pipeline
Fine-Grained Multithreading (II)

- Idea: Switch to another thread every cycle such that no two instructions from a thread are in the pipeline concurrently.

- Tolerates the control and data dependence latencies by overlapping the latency with useful work from other threads.

- Improves pipeline utilization by taking advantage of multiple threads.


Lecture on Fine-Grained Multithreading

Fine-Grained Multithreading

- Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread.
  - Branch/instruction resolution latency overlapped with execution of other threads’ instructions.

+ No logic needed for handling control and data dependences within a thread
  -- Single thread performance suffers
  -- Extra logic for keeping thread contexts
  -- Does not overlap latency if not enough threads to cover the whole pipeline.
Lectures on Fine-Grained Multithreading

  - Pipelined Processor Design (ETH, Spring 2021)
  - https://www.youtube.com/watch?v=6e5KZcCGBYW&list=PL5Q2soXY2Zi_uej3aY39YB5pfW4SJ7LIN&index=16

- Digital Design & Computer Architecture, Spring 2020, Lecture 18c
  - Fine-Grained Multithreading (ETH, Spring 2020)
  - https://www.youtube.com/watch?v=bu5dxKTvQVs&list=PL5Q2soXY2Zi_FRrloMa2fUYWPGiZUBQo2&index=26

https://www.youtube.com/onurmutlulectures
Warps and Warp-Level FGMT

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

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

---

Slide credit: Tor Aamodt
Warp Execution

32-thread warp executing $\text{ADD } A[tid], B[tid] \rightarrow C[tid]$
SIMD Execution Unit Structure

Memory Subsystem

Functional Unit

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 → 4 warps.
Sample GPU SIMT Code (Simplified)

CPU code

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

// there are 100000 threads

CUDA code

```c
// there are 100000 threads
__global__ void KernelFunction(...) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    int varA = aa[tid];
    int varB = bb[tid];
    C[tid] = varA + varB;
}
```
Warps *not* Exposed to GPU Programmers

- CPU threads and GPU kernels
  - Sequential or modestly parallel sections on CPU
  - Massively parallel sections on GPU: Blocks of threads

Serial Code (host)

Parallel Kernel (device)
KernelA\(<\text{<<<} n\text{Blk}, n\text{Thr}\text{>>>}(\text{args});

Serial Code (host)

Parallel Kernel (device)
KernelB\(<\text{<<<} n\text{Blk}, n\text{Thr}\text{>>>}(\text{args});

Slide credit: Hwu & Kirk
Amdahl’s Law

- Amdahl’s Law
  - \( f \): Parallelizable fraction of a program
  - \( N \): Number of processors

\[
\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}
\]


- Maximum speedup limited by serial portion: Serial bottleneck

- All parallel machines “suffer from” the serial bottleneck
Warps not Exposed to GPU Programmers

- CPU threads and GPU kernels
  - Sequential or modestly parallel sections on CPU
  - Massively parallel sections on GPU: Blocks of threads

Serial Code (host)

Parallel Kernel (device)
KernelA<<<nBlk, nThr>>>(args);

Serial Code (host)

Parallel Kernel (device)
KernelB<<<nBlk, nThr>>>(args);

Slide credit: Hwu & Kirk
From Blocks to Warps

- GPU cores: SIMD pipelines
  - Streaming Multiprocessors (SM)
  - Streaming Processors (SP)

- Blocks are divided into warps
  - SIMD unit (32 threads)

NVIDIA Fermi architecture
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
Evolution of NVIDIA GPUs

![Graph showing the evolution of NVIDIA GPUs with increased GFLOPS and functional units over time.](image-url)
108 cores on the A100
(Up to 128 cores in the full-blown chip)

40MB L2 cache

https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/
NVIDIA A100 Core

19.5 TFLOPS Single Precision
9.7 TFLOPS Double Precision
312 TFLOPS for Deep Learning (Tensor cores)

https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/
Tensor Core Microarchitecture (Volta)

- Each warp utilizes two tensor cores
- Each tensor core contains two “octets”
  - 16 SIMD units per tensor core (8 per octet)
  - 4x4 matrix-multiply and accumulate each cycle per tensor core

Unlike conventional SIMD, register contents are not private to each thread, but shared inside the warp

Lecture on Graphics Processing Units

Dynamic Warp Formation Example

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

- **B**
  - Execution of Warp y at Basic Block A

- **C**
  - x/1110 y/0011
  - x/1000 y/0010

- **D**
  - x/0110 y/0001
  - x/0001 y/1100

- **E**
  - x/1110 y/0011

- **F**
  - x/1111 y/1111

- **G**
  - x/1111 y/1111

Baseline

Dynamic Warp Formation

Time

https://youtu.be/eaxGCv0wRrU
Lecture on SIMD Processing & GPUs

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

https://youtu.be/f3IU9IX990o
P&S Heterogeneous Systems

SIMD Processing and GPUs

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich
Spring 2022
22 March 2022
## Clarification of Some GPU Terms

<table>
<thead>
<tr>
<th>Generic Term</th>
<th>NVIDIA Term</th>
<th>AMD Term</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector length</td>
<td>Warp size</td>
<td>Wavefront size</td>
<td>Number of threads that run in parallel (lock-step) on a SIMD functional unit</td>
</tr>
<tr>
<td>Pipelined functional unit / Scalar pipeline</td>
<td>Streaming processor / CUDA core</td>
<td>-</td>
<td>Functional unit that executes instructions for one GPU thread</td>
</tr>
<tr>
<td>SIMD functional unit / SIMD pipeline</td>
<td>Group of N streaming processors (e.g., N=8 in GTX 285, N=16 in Fermi)</td>
<td>Vector ALU</td>
<td>SIMD functional unit that executes instructions for an entire warp</td>
</tr>
<tr>
<td>GPU core</td>
<td>Streaming multiprocessor</td>
<td>Compute unit</td>
<td>It contains one or more warp schedulers and one or several SIMD pipelines</td>
</tr>
</tbody>
</table>