Other Execution Paradigms

- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Systolic Arrays
- Decoupled Access Execute
- SIMD Processing (Vector and Array processors)
- Graphics Processing Units (GPUs)
Readings for this Week

- **Required**

- **Recommended**
Exploiting Data Parallelism: SIMD Processors and GPUs
SIMD Processing:
Exploiting Regular (Data) Parallelism
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
Recall: 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)
Recall: Array vs. Vector Processors

**ARRAY PROCESSOR**

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

**VECTOR PROCESSOR**

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

Instruction Stream

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

Same op @ same time

Different ops @ same space

Different ops @ time

Same op @ space

Space

Time
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 of Memory Banks and Accesses]

Picture credit: Derek Chiou
Recall: Vector Instruction Execution

VADD A, B → C

Execution using one pipelined functional unit


Execution using four pipelined functional units


Time

Space
Recall: Vector Unit Structure

Partitioned Vector Registers

Lane

Functional Unit

Elements 0, 4, 8, ...

Elements 1, 5, 9, ...

Elements 2, 6, 10, ...

Elements 3, 7, 11, ...

Memory Subsystem
Recall: 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

Slide credit: Krste Asanovic
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

Scalar Sequential Code

Vectorized Code

Vectorization is a compile-time reordering of operation sequencing
⇒ requires extensive loop dependence analysis

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

```
padd8 $s2, $s0, $s1
```

<table>
<thead>
<tr>
<th>Bit position</th>
<th>$s0</th>
<th>$s1</th>
<th>$s2</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>a3</td>
<td>b3</td>
<td>a3 + b3</td>
</tr>
<tr>
<td>24 23</td>
<td>a2</td>
<td>b2</td>
<td>a2 + b2</td>
</tr>
<tr>
<td>16 15</td>
<td>a1</td>
<td>b1</td>
<td>a1 + b1</td>
</tr>
<tr>
<td>8 7</td>
<td>a0</td>
<td>b0</td>
<td>a0 + b0</td>
</tr>
</tbody>
</table>

Example:
- $s0 = a3a2a1a0$
- $s1 = b3b2b1b0$
- $s2 = a3b3a2b2a1b1a0b0$

```
0781516232432
```

```
padd8 $s2, $s0, $s1
```
Intel Pentium MMX Operations

- **Idea:** One instruction operates on multiple data elements simultaneously
  - *À la* 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 $x$ on top of the background in image $y$

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

**Figure 9. Generating the selection bit mask.**

MMX Example: Image Overlaying (II)

Y = Blossom image
X = Woman’s image

PAND MM4, MM1

PANDN MM1, MM3

POR MM4, MM1

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

```
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.

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)
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

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

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

**A MIMD Machine with SIMD Processors (II)**

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

More on the Cerebras WSE

https://www.youtube.com/watch?v=x2-qB0J7KHz
Fine-Grained Multithreading
Fine-Grained Multithreading

- Idea: Fetch from a different thread every cycle such that no two instructions from a thread are in the pipeline concurrently
  - Hardware has multiple thread contexts (PC+registers per thread)
  - Threads are completely independent
  - No instruction is fetched from the same thread until the prior branch/instruction from the thread completes

+ No logic needed for handling control and data dependences within a thread
+ High thread-level throughput

-- Single thread performance suffers
-- Extra logic for keeping thread contexts
-- Throughput loss when there are not enough threads to keep the pipeline full

Each pipeline stage has an instruction from a different, completely-independent thread
Fine-Grained Multithreading: Basic Idea

Each pipeline stage has an instruction from a different, completely-independent thread

We need a PC and a register file for each thread + muxes and control
Fine-Grained Multithreading (II)

- **Idea:** Fetch from a different thread every cycle such that no two instructions from a thread are in the pipeline concurrently.

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

- **Improves pipeline utilization** by taking advantage of multiple threads.

- **Improves thread-level throughput** but sacrifices per-thread throughput & latency.

Sun Niagara Multithreaded Pipeline

Fine-Grained Multithreading

- **Advantages**
  + No need for dependence checking between instructions (only one instruction in pipeline from a single thread)
  + No need for branch prediction logic
  + Otherwise-bubble cycles used for executing useful instructions from different threads
  + Improved system throughput, latency tolerance, pipeline utilization

- **Disadvantages**
  - Extra hardware complexity: multiple hardware contexts (PCs, register files, ...), thread selection logic
  - Reduced single thread performance (one instruction fetched every N cycles from the same thread)
  - Resource contention between threads in caches and memory
  - Dependence checking logic between threads may be needed (load/store)
Lecture on Fine-Grained Multithreading

Idea: Fetch from a different thread every cycle such that no two instructions from a thread are in the pipeline concurrently
- Hardware has multiple thread contexts (PC+registers per thread)
- Threads are completely independent
- No instruction is fetched from the same thread until the prior branch/instruction from the thread completes

+ No logic needed for handling control and data dependences within a thread
+ High thread-level throughput
  -- Single thread performance suffers
  -- Extra logic for keeping thread contexts
  -- Throughput loss when there are not enough threads to keep the pipeline full

Each pipeline stage has an instruction from a different, completely-independent thread

Digital Design & Computer Architecture - Lecture 14: Pipelined Processor Design (Spring 2022)
1,066 views • Streamed live on Apr 8, 2022

Onur Mutlu
24.5K subscribers

Digital Design and Computer Architecture, ETH Zürich, Spring 2022 (https://safari.ethz.ch/digitaltechnik/)

Lecture 14: Pipelined Processor Design
Lecturer: Professor Onur Mutlu (https://people.inf.ethz.ch/omutlu/)
Date: April 8, 2022

https://youtu.be/XaW_O9nKPe0?t=5070
Lectures on Fine-Grained Multithreading

- Digital Design & Computer Architecture, Spring 2022, Lecture 14
  - Pipelined Processor Design (ETH, Spring 2022)
  - https://youtu.be/XaW_O9nKPe0?t=5070

- Digital Design & Computer Architecture, Spring 2020, Lecture 18c
  - Fine-Grained Multithreading (ETH, Spring 2020)
  - https://www.youtube.com/watch?v=bu5dxKTvQVs&list=PL5Q2soXY2Zi_FRrIoMa2fUYWPgiZUBQo2&index=26
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?

Scalar Sequential Code

```c
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

for (i=0; i < N; i++)
C[i] = A[i] + B[i];
**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 → V1
VLD B → V2
VADD V1 + V2 → V3
VST V3 → 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

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**!
SPMD on SIMT Machine

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

Iter. 1
Iter. 2

Warp: A set of threads that execute the same instruction (i.e., at the same PC)

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:

- Warp 0 at PC X
- Warp 20 at PC X+2
- Iter. 20*32 + 1
- Iter. 20*32 + 2
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 ...

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

Slide credit: Tor Aamodt
Warp Execution (Recall the Slide)

32-thread warp executing ADD A[tid], B[tid] → C[tid]

Execution using one pipelined functional unit

Execution using four pipelined functional units

Slide credit: Krste Asanovic
SIMD Execution Unit Structure

- **Functional Unit**
- **Memory Subsystem**
- **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
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
Sample GPU SIMT Code (Simplified)

CPU code

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

CUDA code

// 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;
}
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);
}
```
Lecture on GPU Programming

Data Reuse: Tiling

To take advantage of data reuse, we divide the input into tiles that can be loaded into shared memory.

```c
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];

Load tile into shared memory
__syncthreads();
for (int i = 0; i < 3; i++){
  for (int j = 0; j < 3; j++){
    sum += gauss[i][j] * l_data[(i+L_row-1)*(L_SIZE+2)+j+L_col-1];
  }
}
```
Heterogeneous Systems Course (Spring 2022)

- Short weekly lectures
- Hands-on projects

https://youtube.com/playlist?list=PL5Q2soXY2Zi9XRgXR38IM_FTjmY6h7Gzm
https://safari.ethz.ch/projects_and_seminars/spring2022/doku.php?id=heterogeneous_systems
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
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

![Diagram showing warp formation and merging]

<table>
<thead>
<tr>
<th>Warp X</th>
<th>Warp Y</th>
<th>Warp Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

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

Baseline

Dynamic Warp Formation
Hardware Constraints Limit Flexibility of Warp Grouping

Can you move any thread flexibly to any lane?

Slide credit: Krste Asanovic
Large Warps and Two-Level Warp Scheduling

- Two main reasons for GPU resources be underutilized
  - Branch divergence
  - Long latency operations

Large Warp Microarchitecture Example

- Reduce **branch divergence** by having large warps
- Dynamically break down a large warp into sub-warsps

Decide Stage

Two-Level Round Robin

- **Scheduling in two levels** to deal with long latency operations

Core: All Warps Compute
Memory System: Req Warp 0, Req Warp 1, Req Warp 15

Round Robin Scheduling, 16 total warps

Core: Group 0 Compute, Group 1 Compute
Memory System: Req Warp 0, Req Warp 1, Req Warp 7

Two Level Round Robin Scheduling, 2 fetch groups, 8 warps each

Large Warps and Two-Level Warp Scheduling

- Veynu Narasiman, Chang Joo Lee, Michael Shebanow, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt,

"Improving GPU Performance via Large Warps and Two-Level Warp Scheduling"

Proceedings of the 44th International Symposium on Microarchitecture (MICRO), Porto Alegre, Brazil, December 2011. Slides (ppt)

---

Improving GPU Performance via Large Warps and Two-Level Warp Scheduling

Veynu Narasiman†, Michael Shebanow‡, Chang Joo Lee¶, Rustam Miftakhutdinov†, Onur Mutlu§, Yale N. Patt†

†The University of Texas at Austin {narasima, rustam, patt}@hps.utexas.edu
‡Nvidia Corporation mshebanow@nvidia.com
¶Intel Corporation chang.joo.lee@intel.com
§Carnegie Mellon University onur@cmu.edu
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

Slide credit: Kayvon Fatahalian
30 cores on the GTX 285: 30,720 threads
Evolution of NVIDIA GPUs
NVIDIA V100

- **NVIDIA-speak:**
  - 5120 stream processors
  - “SIMT execution”

- **Generic speak:**
  - 80 cores
  - 64 SIMD functional units per core
  - Tensor cores for Machine Learning

NVIDIA V100 Block Diagram

80 cores on the V100
NVIDIA V100 Core

15.7 TFLOPS Single Precision
7.8 TFLOPS Double Precision
125 TFLOPS for Deep Learning (Tensor cores)

https://devblogs.nvidia.com/inside-volta/
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

Edge TPU: Baseline Accelerator

**ML Model**

**DRAM**

**Input Activation**

**Parameter**

**Output Activation**

**Dataflow**

**PE Array**

**Buffer**

- 64x64 array
- 2TFLOP/s
- 4MB on-chip buffer

**Input Activation**

**Parameter**

**Output Activation**

**TPU and Model Characterization**

**Mensa Framework**

**Mensa-G**

**Evaluation**

**Conclusion**
Root Cause of Accelerator Challenges

The key components of Google Edge TPU are completely oblivious to layer heterogeneity.

Edge accelerators typically take a monolithic approach: equip the accelerator with an over-provisioned PE array and on-chip buffer, a rigid dataflow, and fixed off-chip bandwidth.
Lecture 19b: Systolic Array Architectures

An Example Modern Systolic Array: TPU (II)

As reading a large SRAM uses much more power than arithmetic, the matrix unit uses systolic execution to save energy by reducing reads and writes of the Unified Buffer [Kun80][Ram91][Ovt15b]. Figure 4 shows that data flows in from the left, and the weights are loaded from the top. A given 256-element multiply-accumulate operation moves through the matrix as a diagonal wavefront. The weights are preloaded, and take effect with the advancing wave alongside the first data of a new block. Control and data are pipelined to give the illusion that the 256 inputs are read at once, and that they instantly update one location of each of 256 accumulators. From a correctness perspective, software is unaware of the systolic nature of the matrix unit, but for performance, it does worry about the latency of the unit.


Digital Design & Computer Arch. - Lecture 19: VLIW and Systolic Array Architectures (Spring 2022)

https://youtu.be/1SSqV7Y75oU?t=2316
NVIDIA A100

- **NVIDIA-speak:**
  - 6912 stream processors
  - “SIMT execution”

- **Generic speak:**
  - 108 cores
  - 64 SIMD functional units per core

- **Tensor cores for Machine Learning**
  - Support for sparsity
  - New floating point data type (TF32)

108 cores on the A100
(Upto 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/
Evolution of NVIDIA GPUs (Updated)
NVIDIA H100 Block Diagram

144 cores on the full GH100
60MB L2 cache

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
NVIDIA H100 Core

48 TFLOPS Single Precision*
24 TFLOPS Double Precision*
800 TFLOPS (FP16, Tensor Cores)*

* Preliminary performance estimates

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
Food for Thought

- Compare and contrast GPUs vs Systolic Arrays
  - Which one is better for machine learning?
  - Which one is better for image/vision processing?
  - What types of parallelism each one exploits?
  - What are the tradeoffs?

- If you are interested in such questions and more...
  - Bachelor’s Seminar in Computer Architecture (HS2022, FS2023)
  - Computer Architecture Master’s Course (HS2022)
Heterogeneous Systems Course (Spring 2022)

- Short weekly lectures
- Hands-on projects

https://youtube.com/playlist?list=PL5Q2soXY2ZlgXR38IM_FTjmY6h7Gzm

https://safari.ethz.ch/projects_and_seminars/spring2022/doku.php?id=heterogeneous_systems
Heterogeneous Systems Course (Fall 2021)

- Short weekly lectures
- Hands-on projects

Livestream - P&S Hands-on Acceleration on Heterogeneous Computing Systems (Fall 2021)

https://youtube.com/playlist?list=PL5Q2soXY2Zi_OwkTgEyA6tk3UsoPBH737

https://safari.ethz.ch/projects_and_seminars/fall2021/doku.php?id=heterogeneous_systems
# 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>