Digital Design & Computer Arch.

Lecture 21: Graphics Processing Units

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich
Spring 2022
13 May 2022

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)

Problem
Algorithm
Program/Language
System Software
SW/HW Interface
Micro-architecture
Logic
Devices
Electrons

Readings for this Week

- **Required**
  - Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.

- **Recommended**
  - Peleg and Weiser, “MMX Technology Extension to the Intel Architecture,” IEEE Micro 1996.

Exploiting Data Parallelism: SIMD Processors and GPUs

SIMD Processing:
Exploiting Regular (Data) Parallelism

Recall: Flynn's Taxonomy of Computers

- Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966

- 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

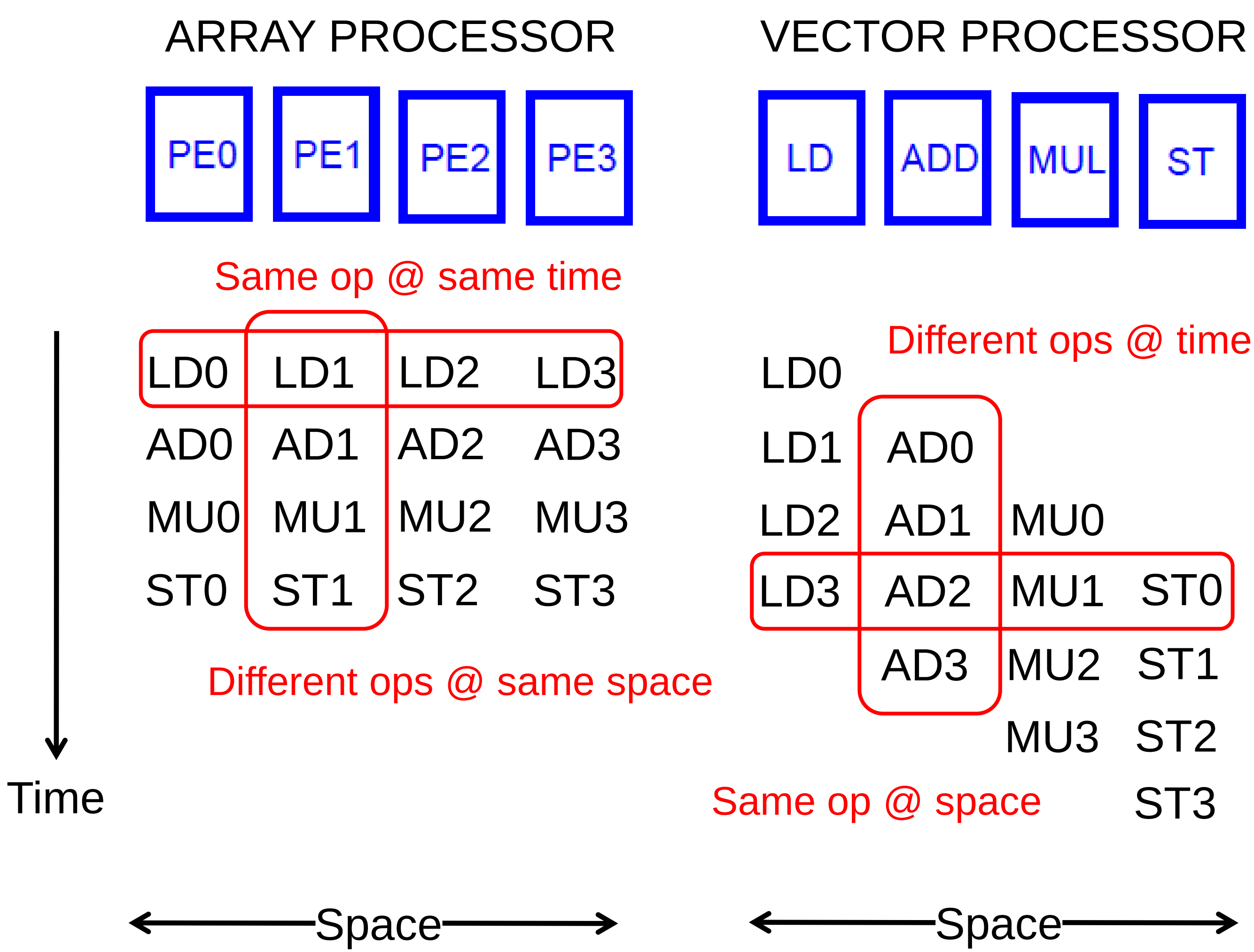
  - Array processor: Instruction operates on multiple data elements at the same time using different spaces (PEs)

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

Recall: Array vs. Vector Processors

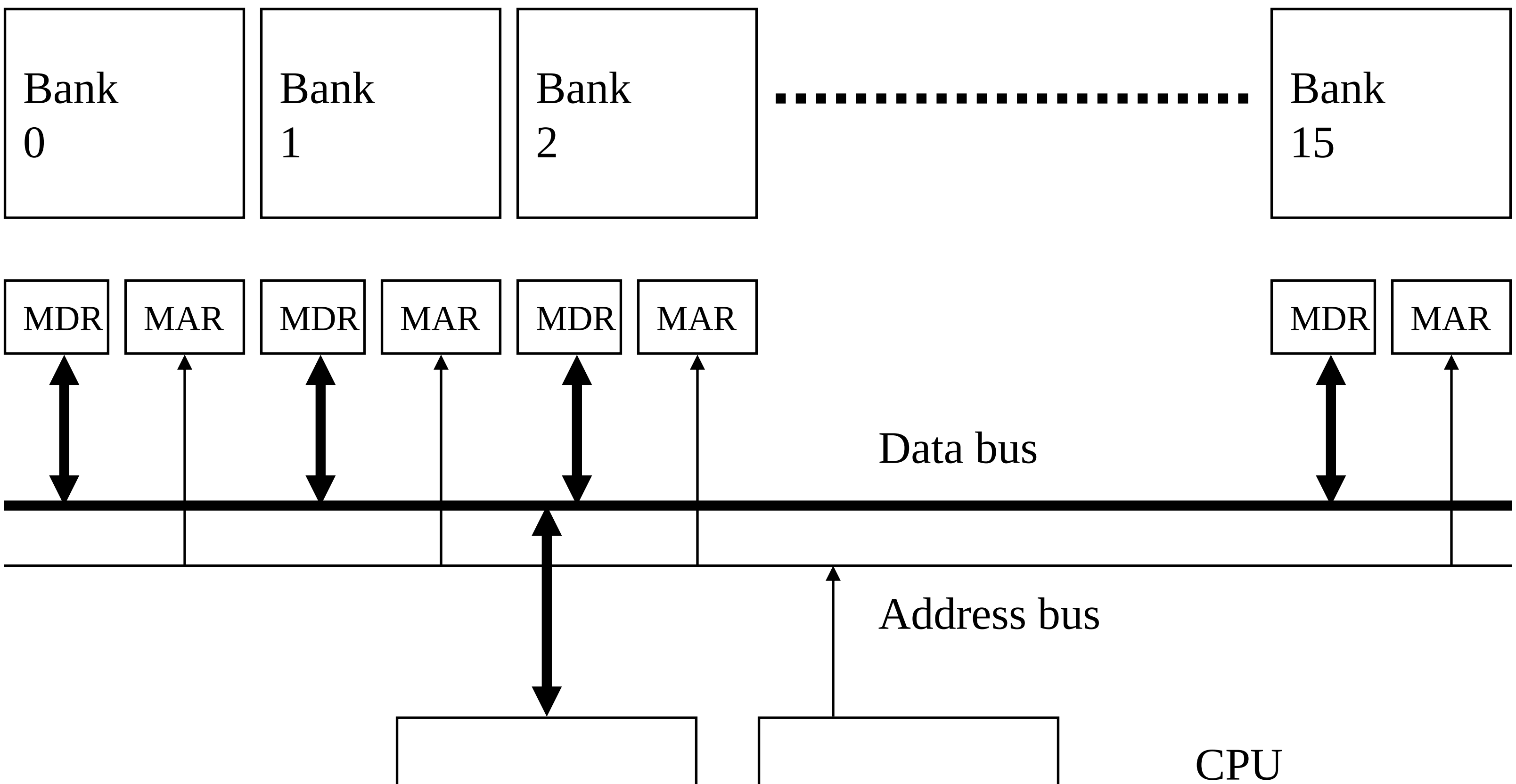
Instruction Stream

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



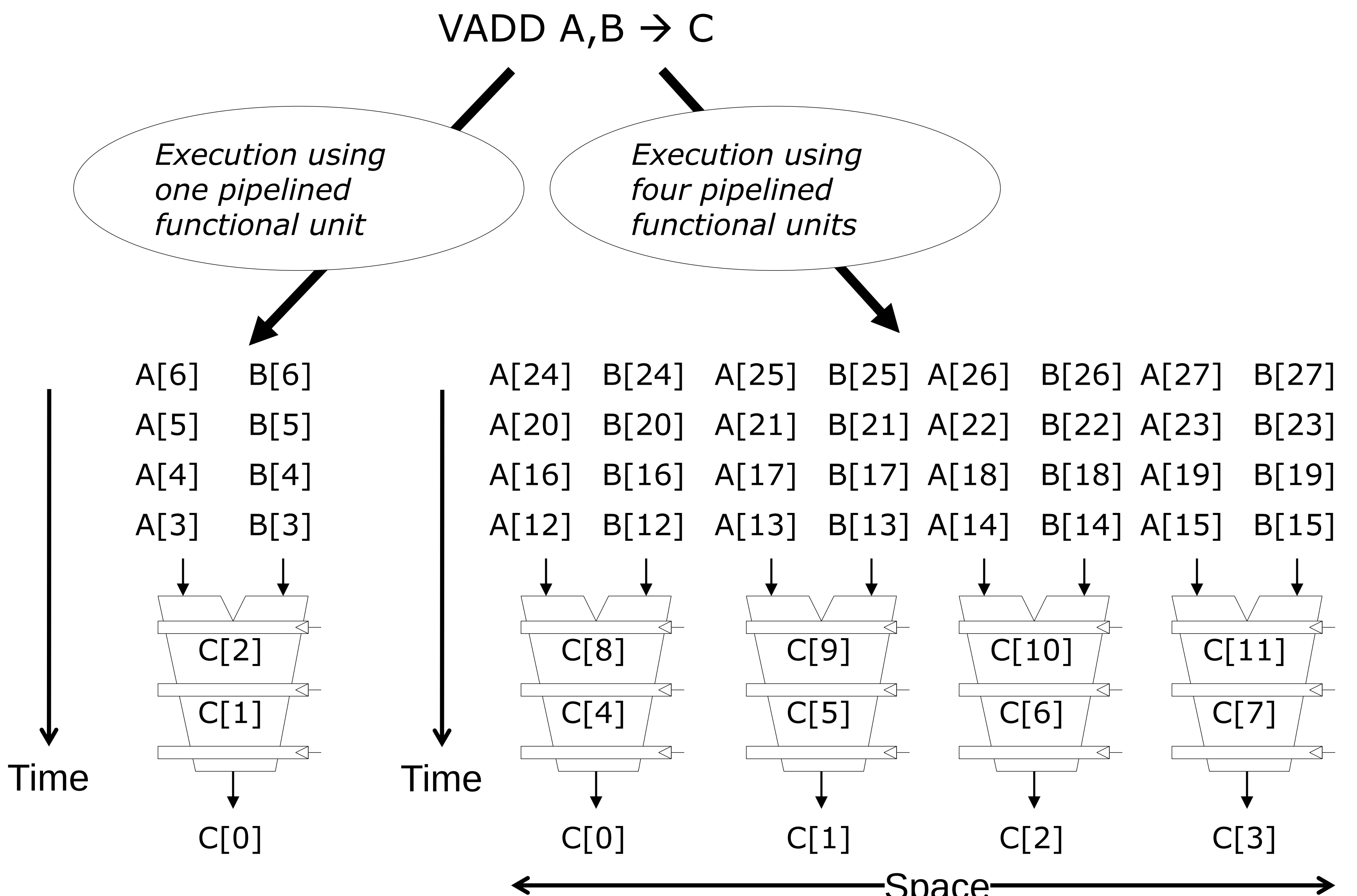
Recall: Memory Banking

- 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

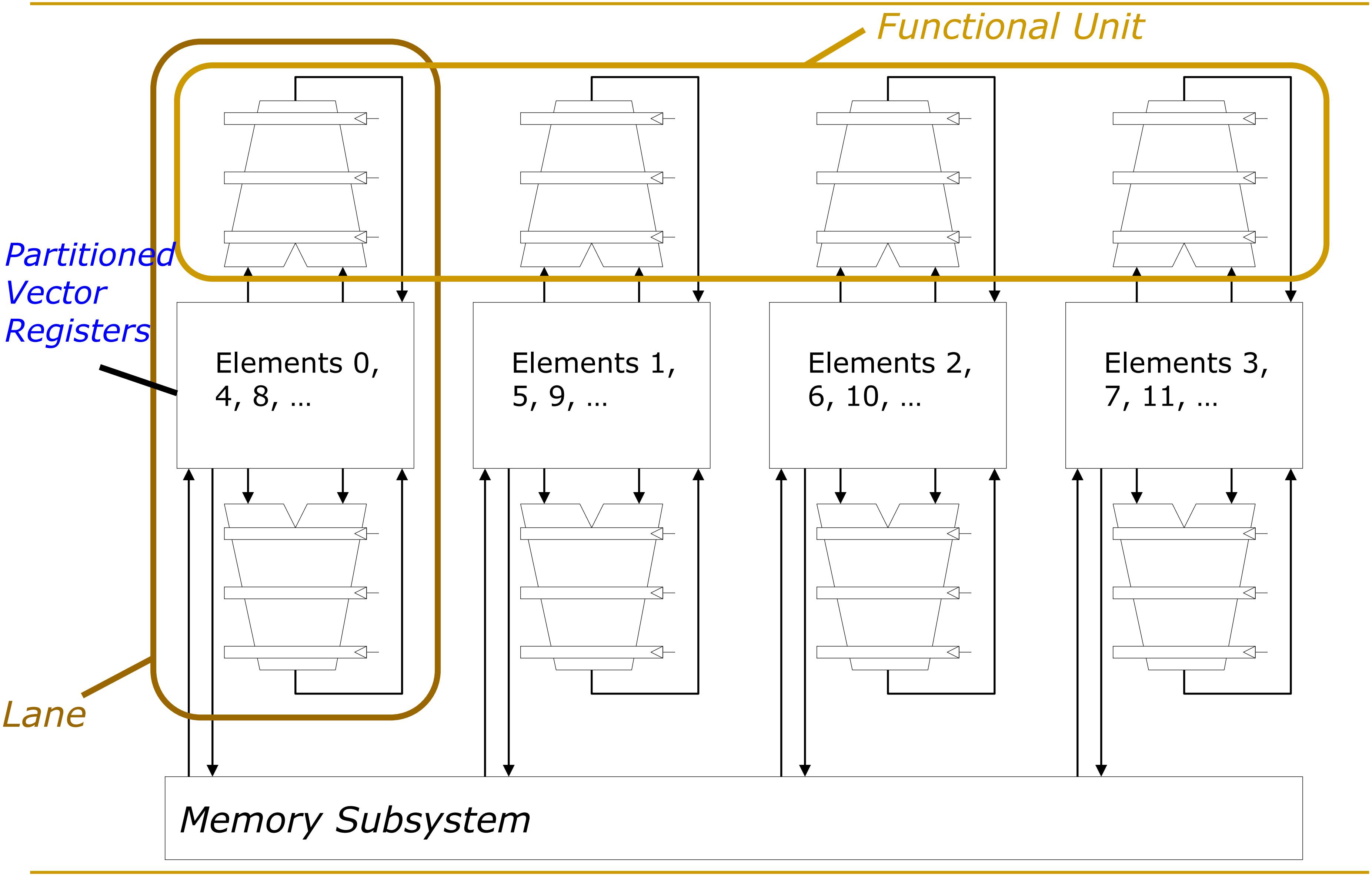


Picture credit: Derek Chiou

Recall: Vector Instruction Execution



Recall: Vector Unit Structure

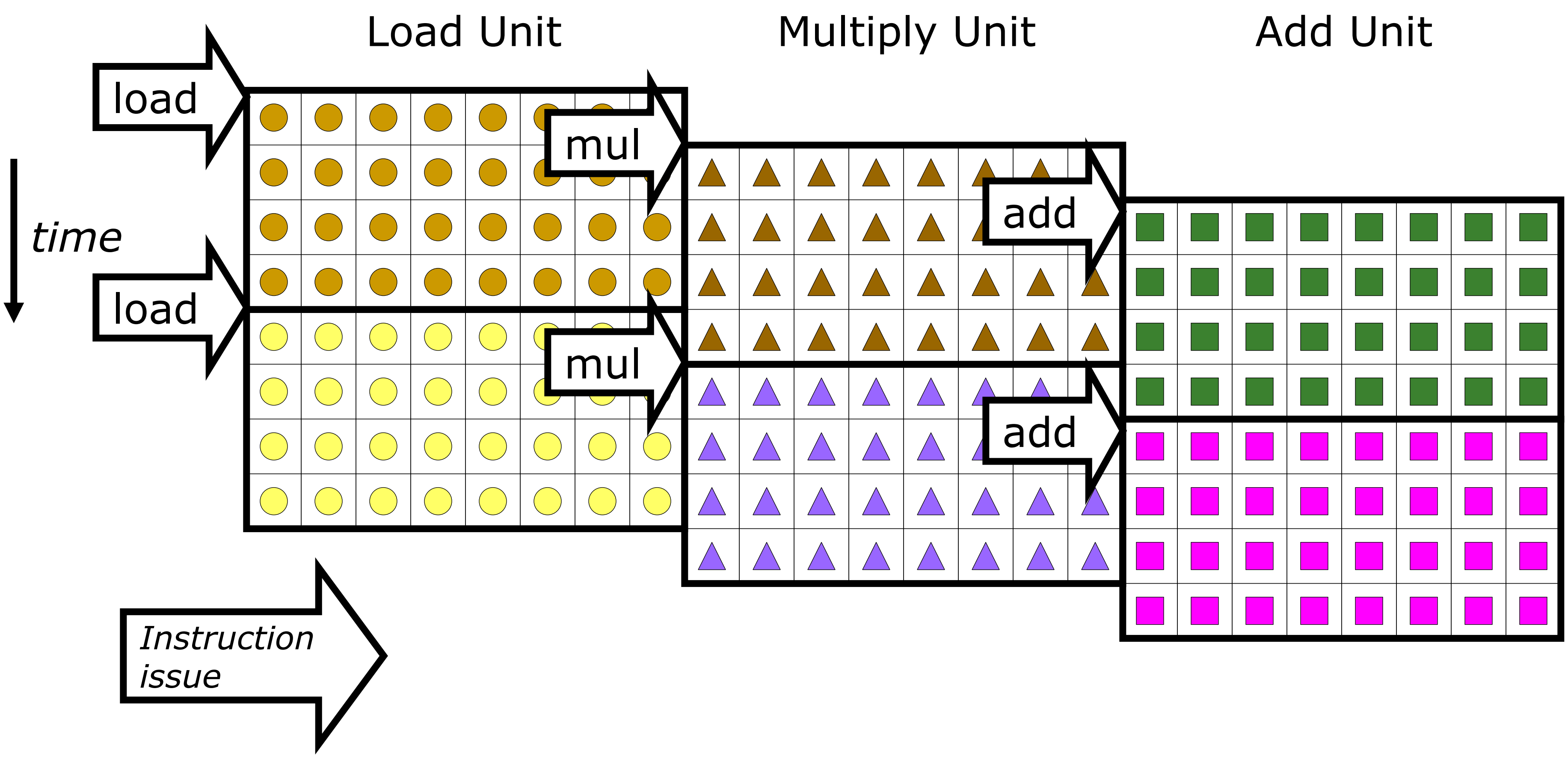


Slide credit: Krste Asanovic

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



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?

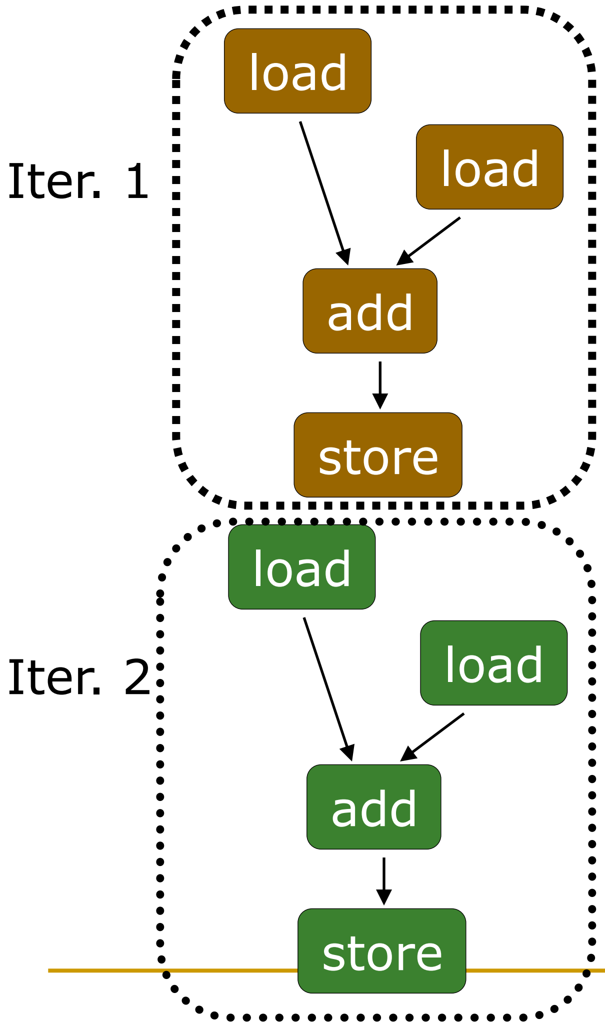
To program a vector machine, the compiler or hand coder must make the data structures in the code fit nearly exactly the regular structure built into the hardware. That's hard to do in first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated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

Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983.

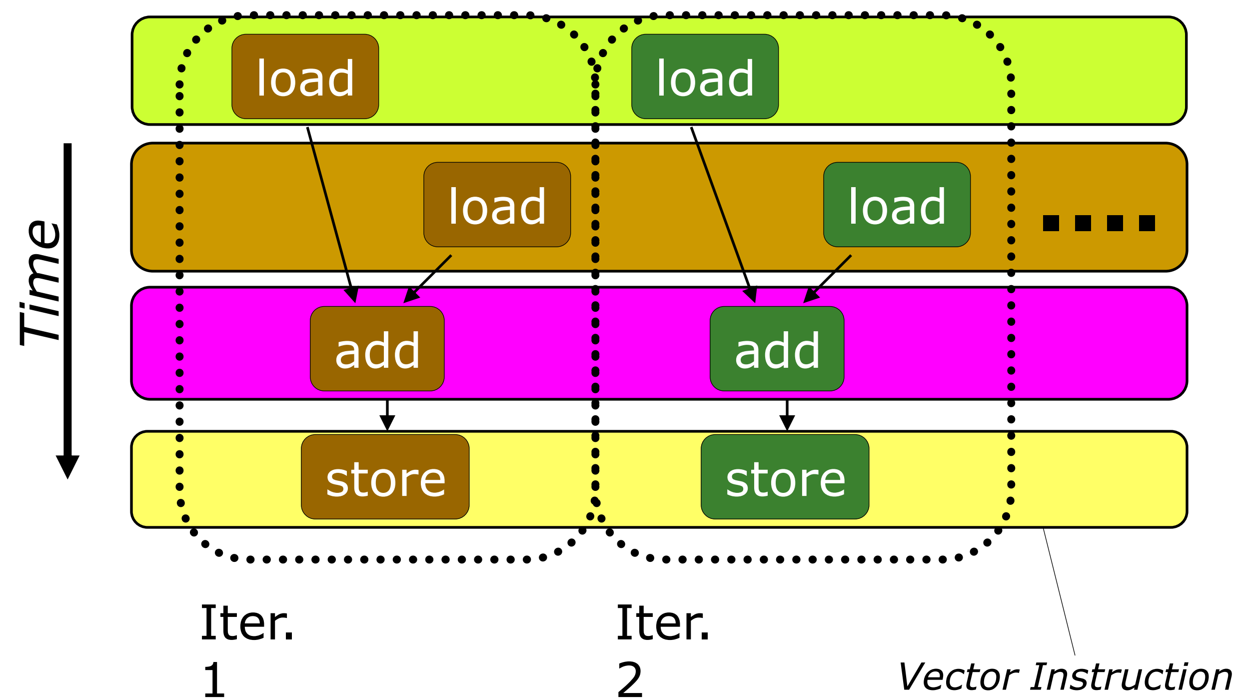
Automatic Code Vectorization

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

Scalar Sequential Code



Vectorized Code



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

Slide credit: Krste Asanovic

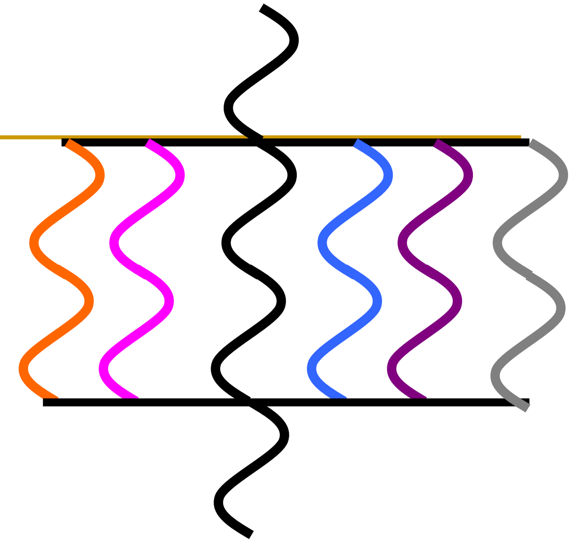
Vector/SIMD Processing Summary

- 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

Recall: Amdahl's Law



- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors

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

  - Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

- Maximum speedup limited by serial portion: Serial bottleneck

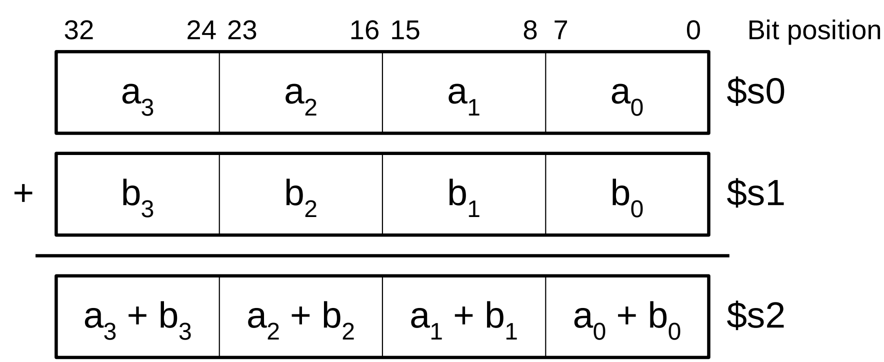
- All parallel machines "suffer from" the serial bottleneck

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
```



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

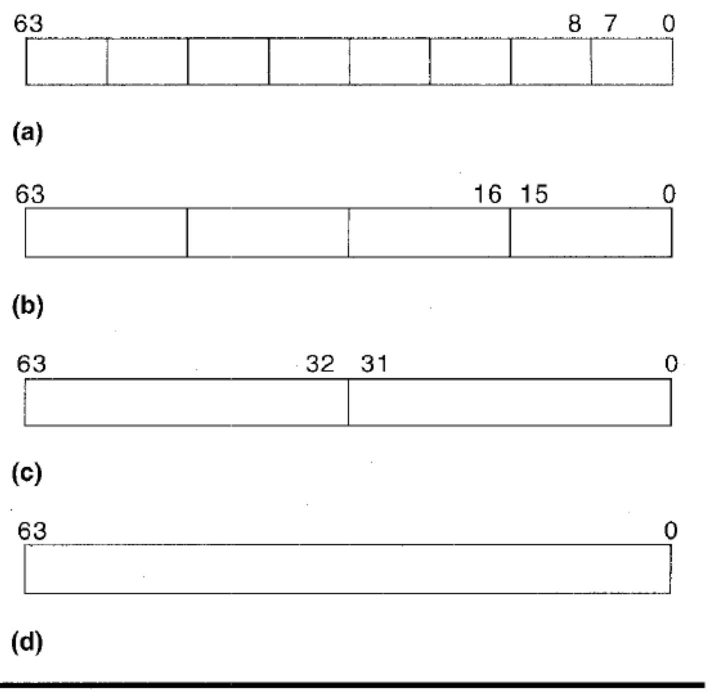


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

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.

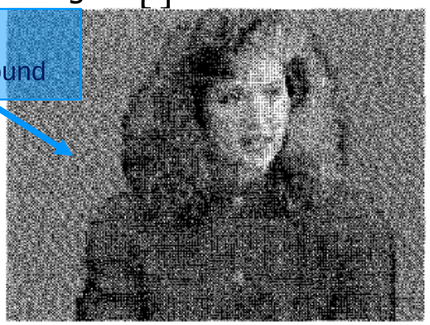
Peleg and Weiser, "MMX Technology Extension to the Intel Architecture," IEEE Micro, 1996.

MMX Example: Image Overlaying (I)

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

Image x[ ]

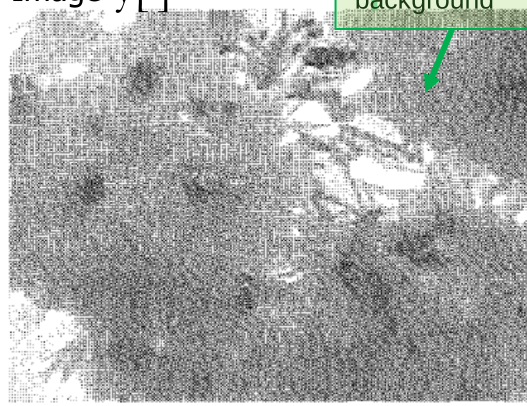
Blue background



+

Image y[ ]

Blossom background



=

Image new_image[ ]

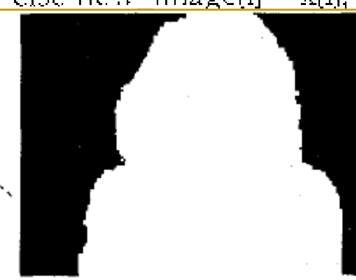
code operation is

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

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

PCMPEQB MM1, MM3

	MM1	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue
Image x[ ]	MM3	X7!=blue	X6!=blue	X5=blue	X4=blue	X3!=blue	X2!=blue	X1=blue	X0=blue
Bit mask	MM1	0x0000	0x0000	0xFFFF	0xFFFF	0x0000	0x0000	0xFFFF	0xFFFF

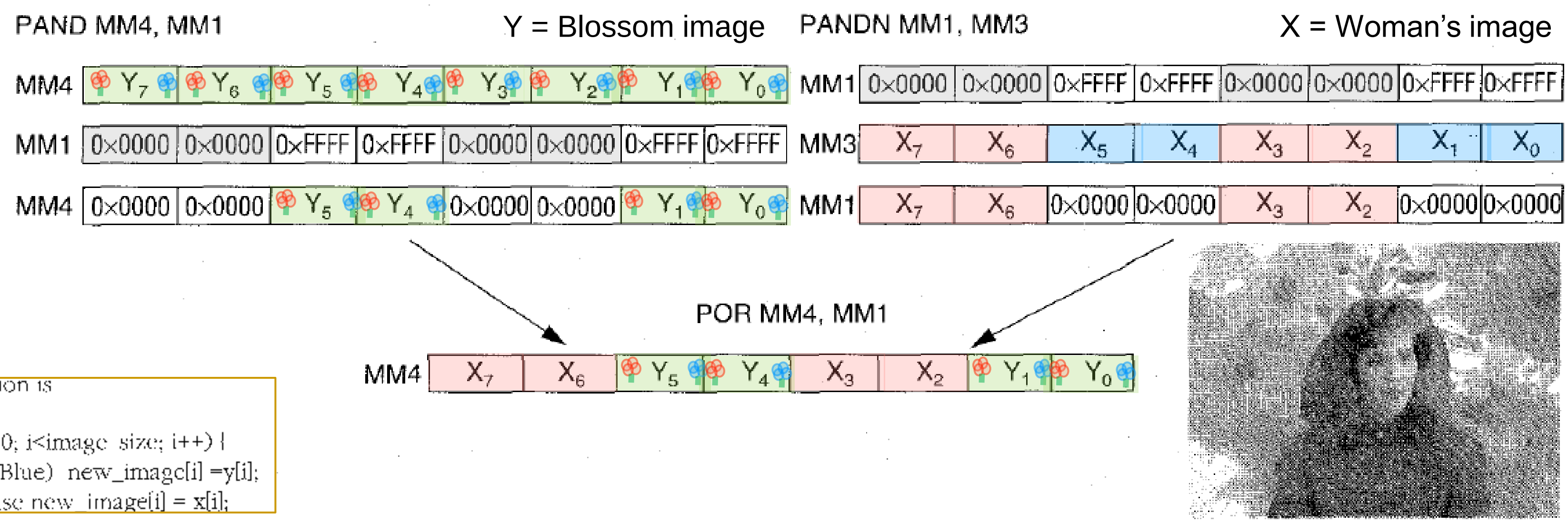


Bitmask

Figure 9. Generating the selection bit mask.

Peleg and Weiser, "MMX Technology Extension to the Intel Architecture," IEEE Micro, 1996.

MMX Example: Image Overlaying (II)



code operation is

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

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

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

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

Peleg and Weiser, "MMX Technology Extension to the Intel Architecture," IEEE Micro, 1996.

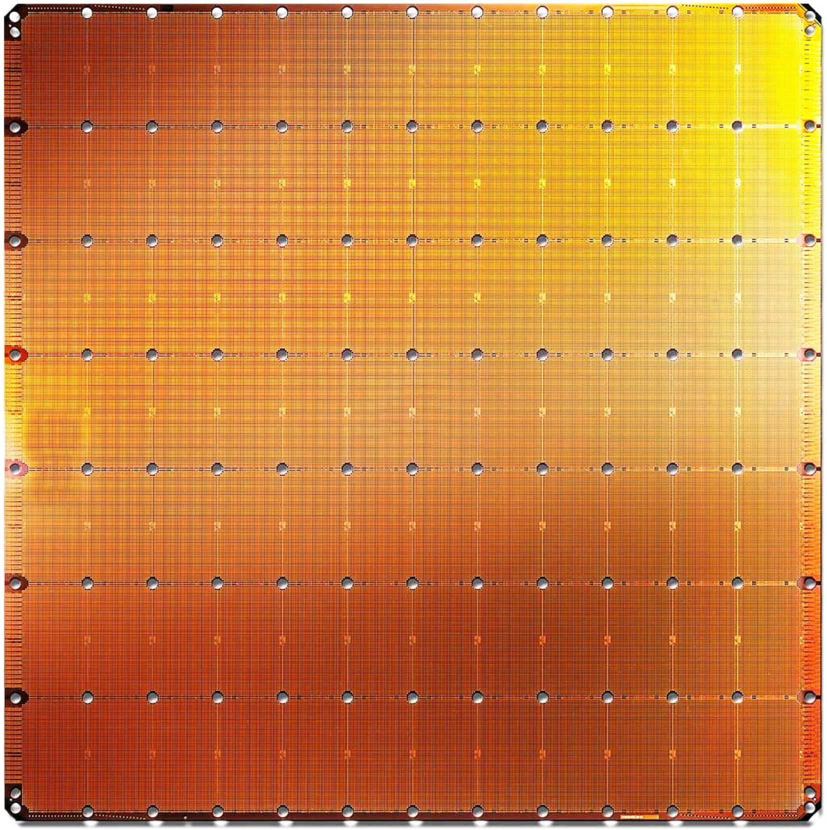
From MMX to AMX in x86 ISA

- 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)

https://www.intel.sg/content/dam/www/public/apac/xa/en/pdfs/ssg/Intel_Processor_Architecture_SIMD_Instructions.pdf
https://gcc.gnu.org/wiki/cauldron2014?action=AttachFile&do=get&target=Cauldron14_AVX-512_Vector_ISA_Kirill_Yukhin_20140711.pdf

SIMD Operations in Modern (Machine Learning) Accelerators

Cerebras's Wafer Scale Engine (2019)



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

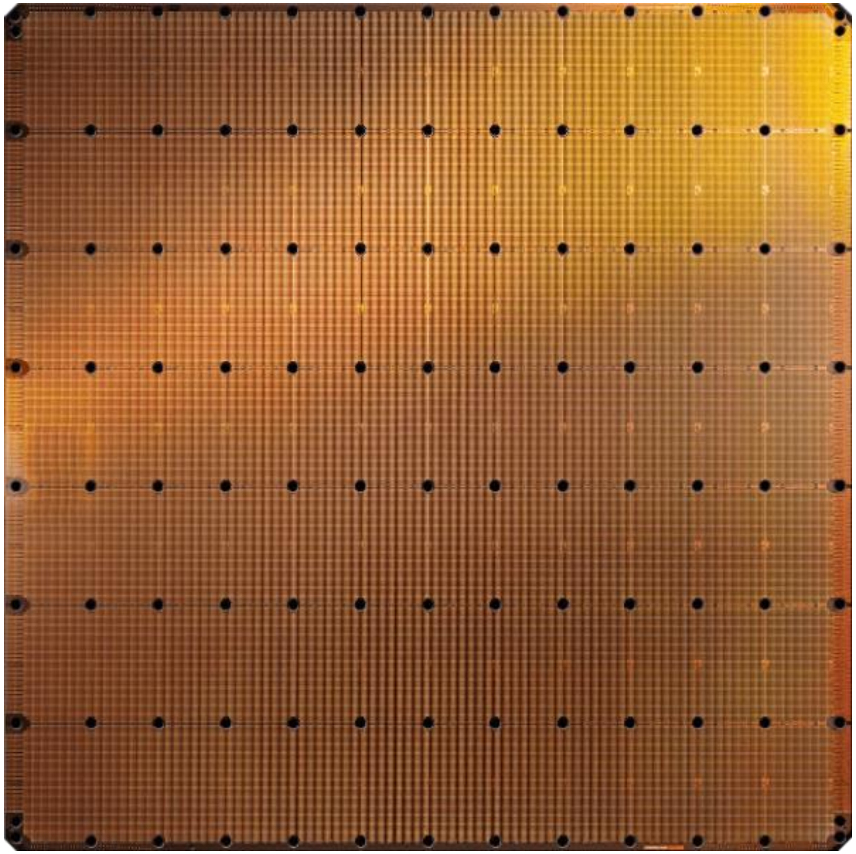
Cerebras WSE
1.2 Trillion transistors
46,225 mm^2

Largest GPU
21.1 Billion transistors
815 mm^2
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 mm^2

Largest GPU
54.2 Billion transistors
826 mm^2
NVIDIA Ampere GA100

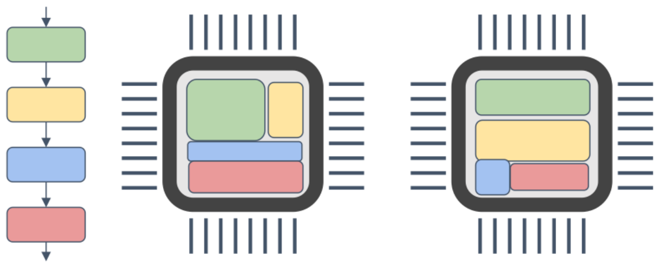
https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning

https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning/

Size, Place, and Route in Cerebras's WSE

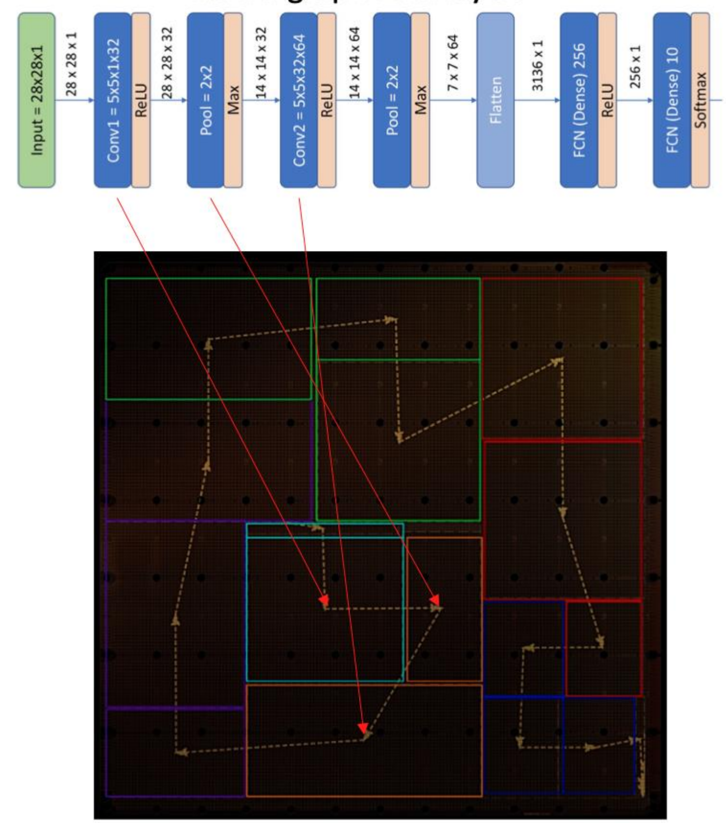
- Neural network mapping onto the whole wafer is a challenge

Multiple possible mappings



An example mapping

Kernel graph with layers



Layers mapped on Wafer Scale Engine

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."

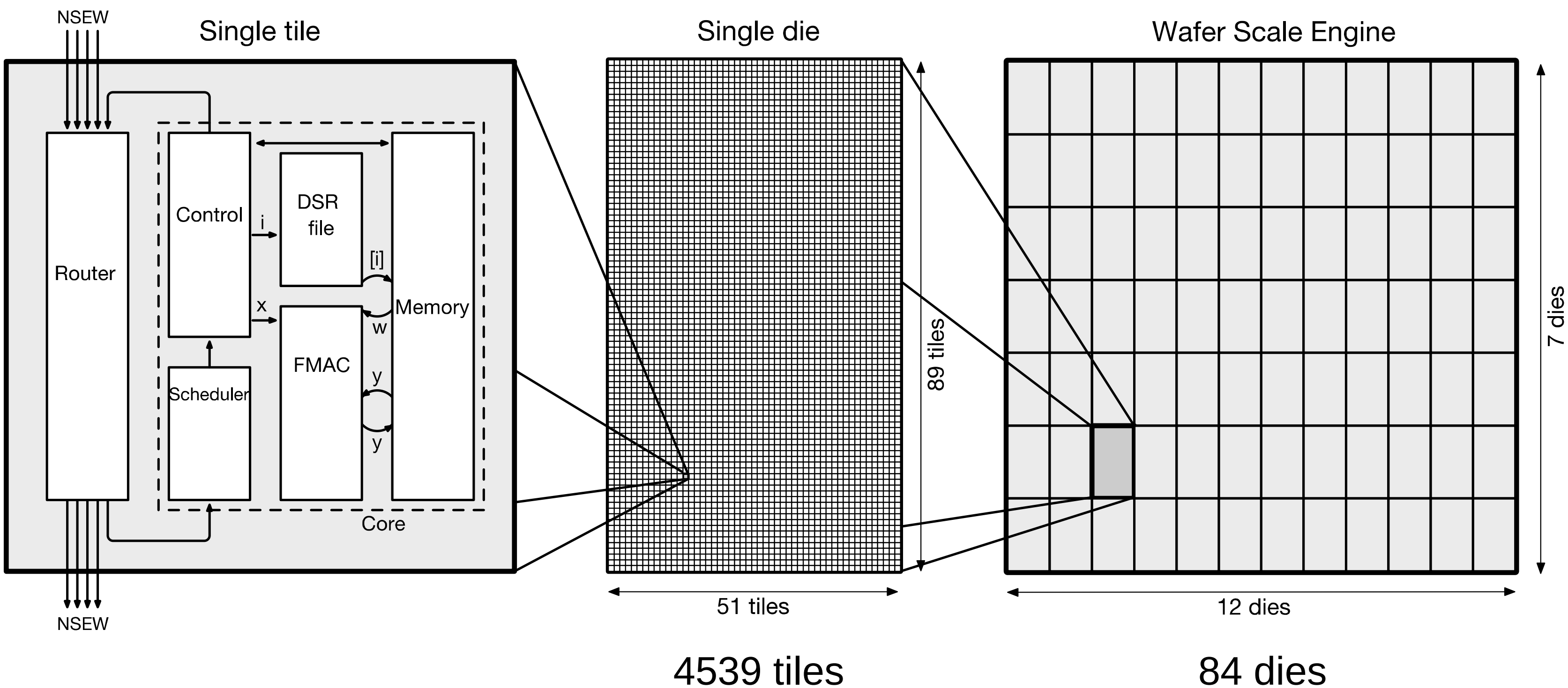
Recall: Flynn's Taxonomy of Computers

- Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966

- 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)

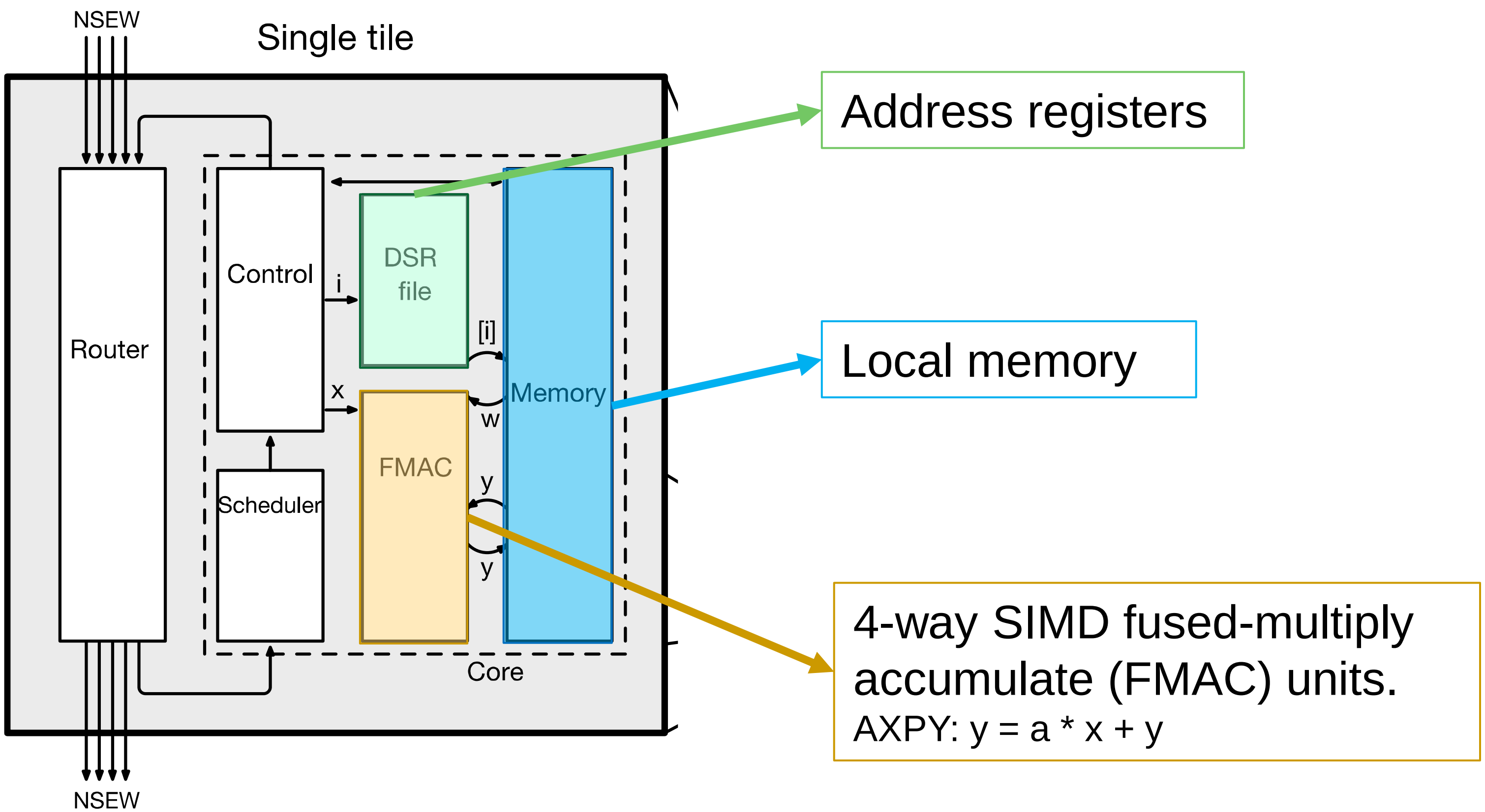
- MIMD machine
  - Distributed memory (no shared memory)
  - 2D-mesh interconnection fabric



Rocki et al., “Fast stencil-code computation on a wafer-scale processor.” SC 2020.

A MIMD Machine with SIMD Processors (II)

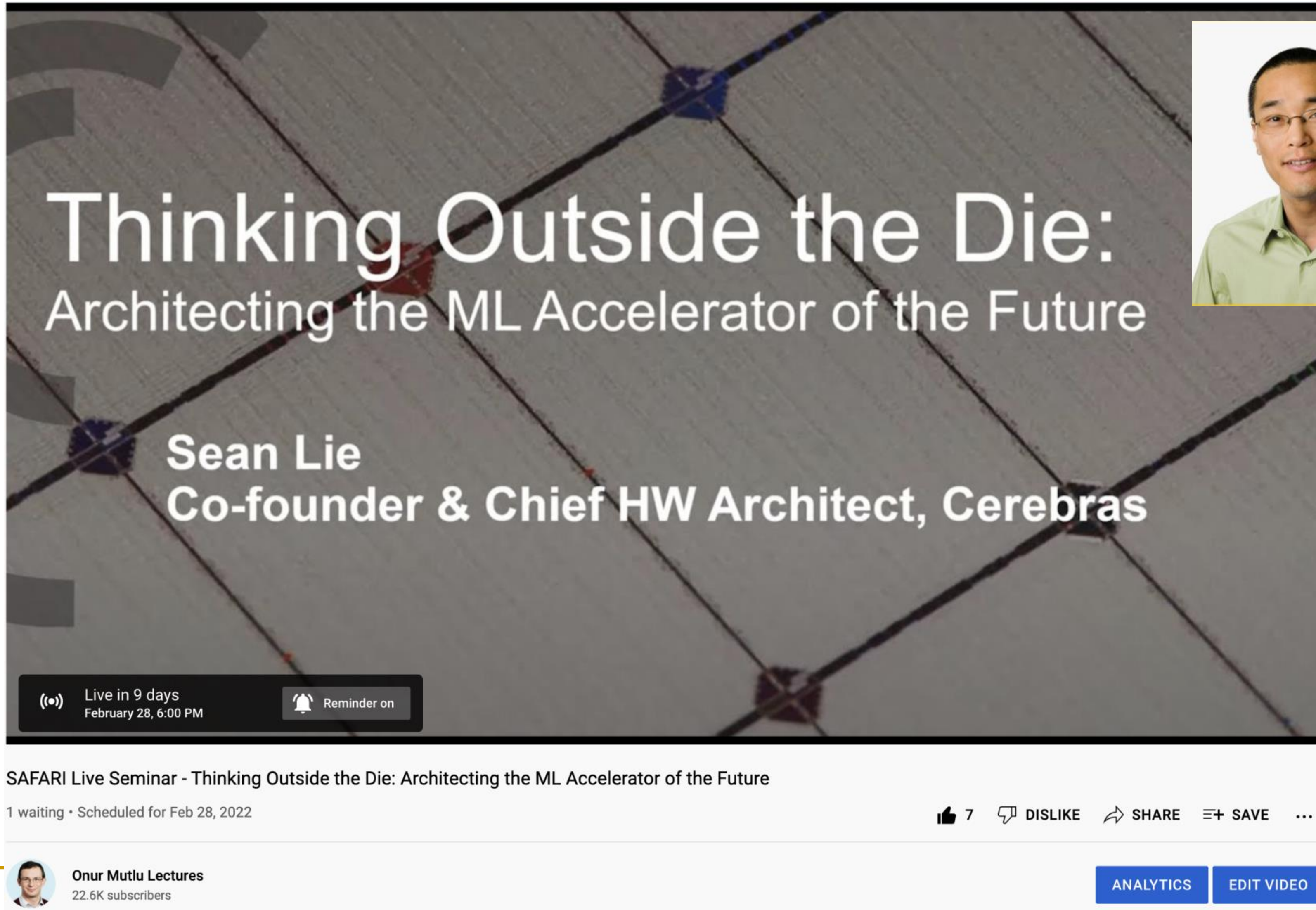
- SIMD processors
  - 4-way SIMD for 16-bit floating point operands
  - 48 KB of local SRAM



Rocki et al., “Fast stencil-code computation on a wafer-scale processor.” SC 2020.

More on the Cerebras WSE

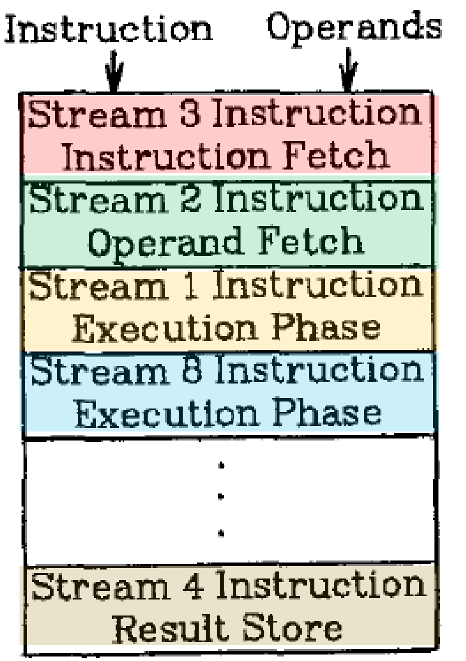
https://www.youtube.com/watch?v=x2-qB0J7KHw



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

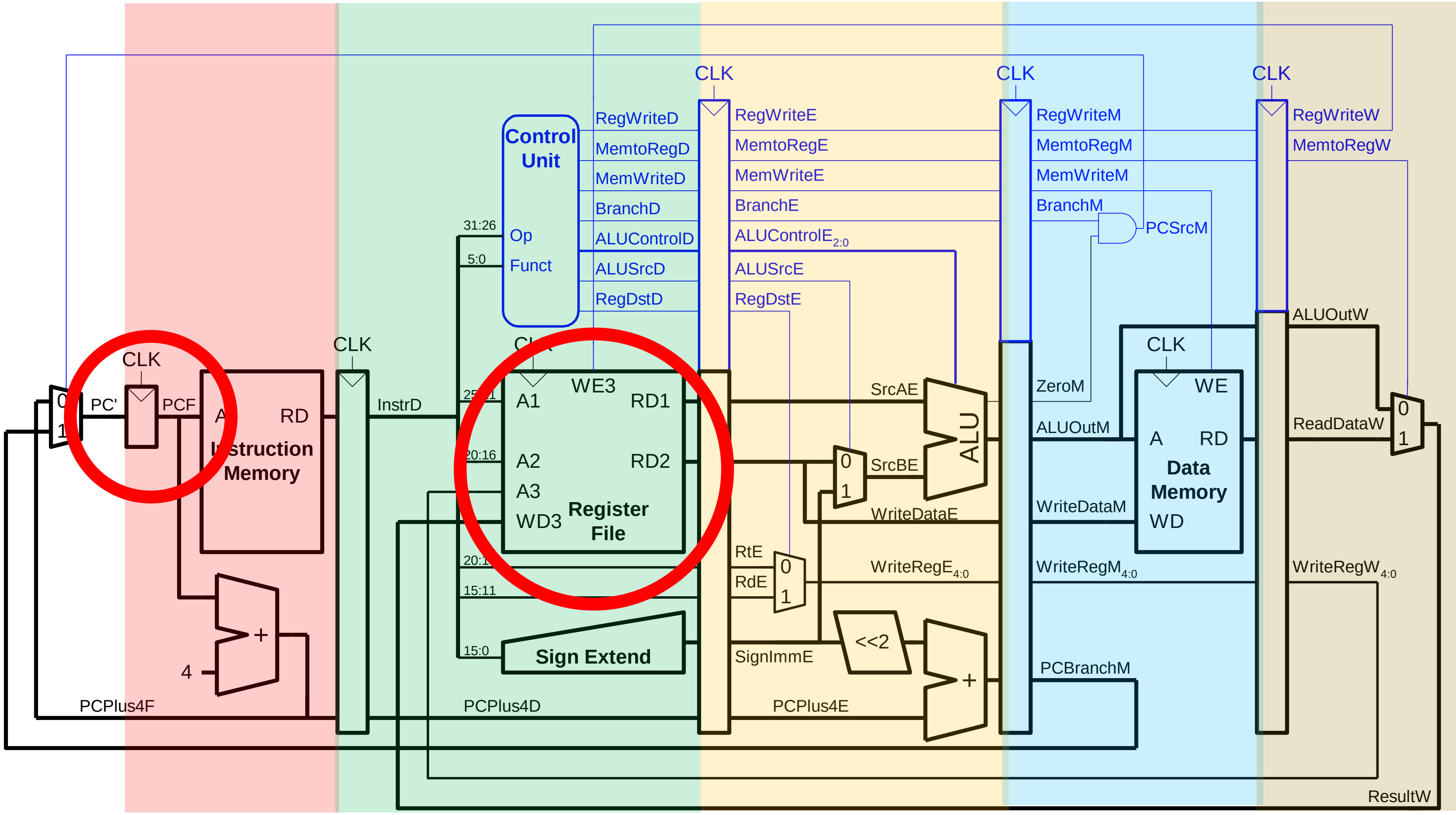
-- Single thread performance suffers

-- 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

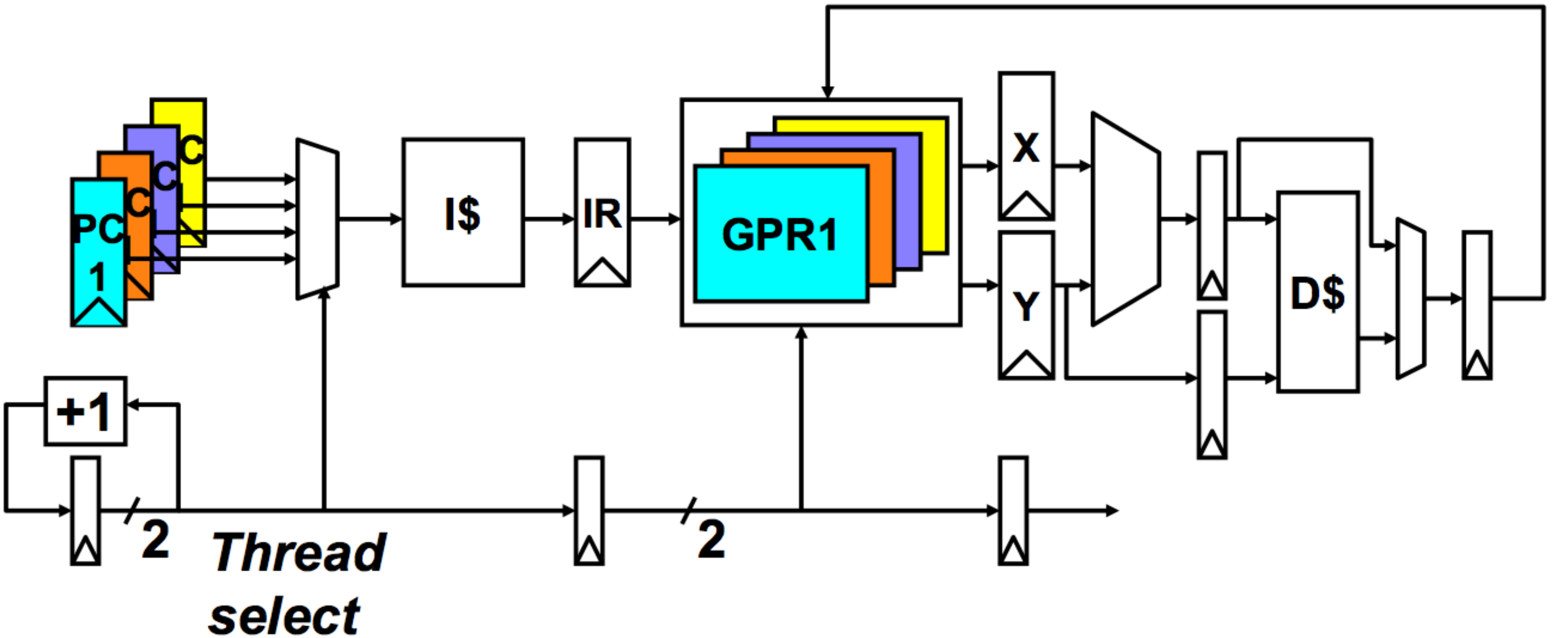
Fine-Grained Multithreading (II)

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

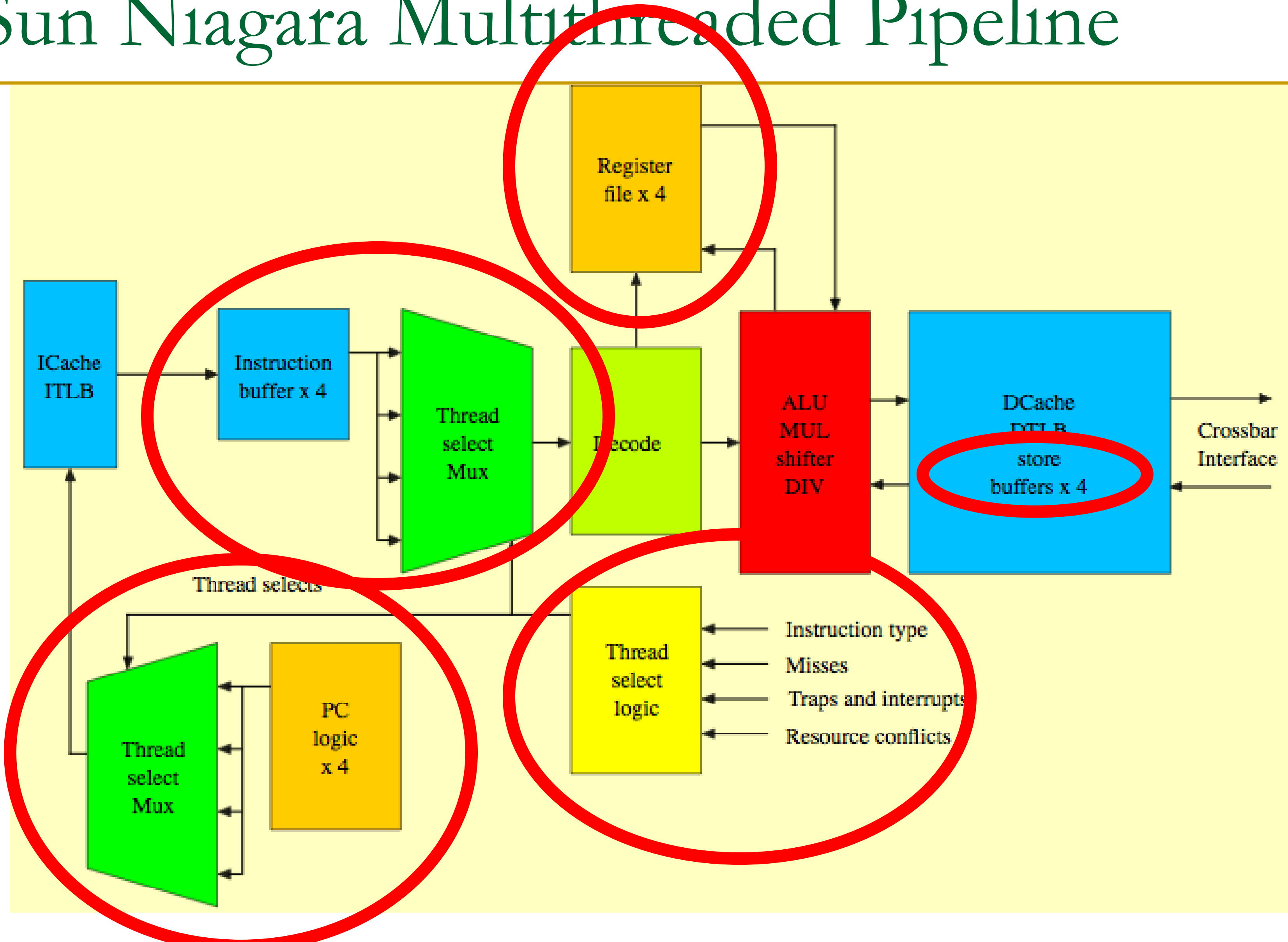
- Tolerates control and data dependence resolution latencies by overlapping the latency with useful work from other threads
- Improves pipeline utilization by taking advantage of multiple threads
- Improves thread-level throughput but sacrifices per-thread throughput & latency

- Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.
- Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.

Multithreaded Pipeline Example



Sun Niagara Multithreaded Pipeline



Kongetira et al., "Niagara: A 32-Way Multithreaded Sparc Processor," IEEE Micro 2005.

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

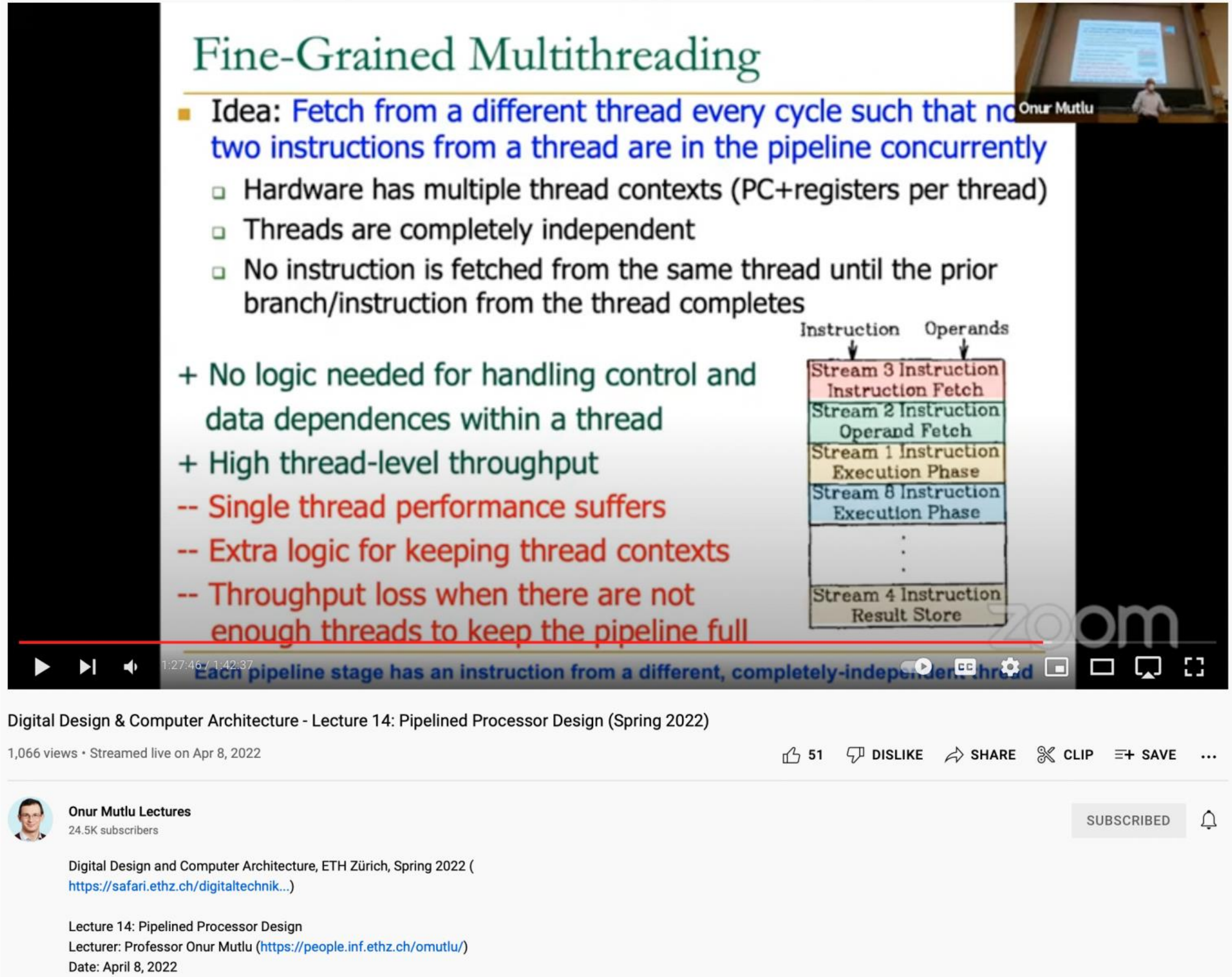
  \- Extra hardware complexity: multiple hardware contexts (PCs, register files, …), thread selection logic

  \- 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



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_FRrloMa2fUYWPGiZUBQo2&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)

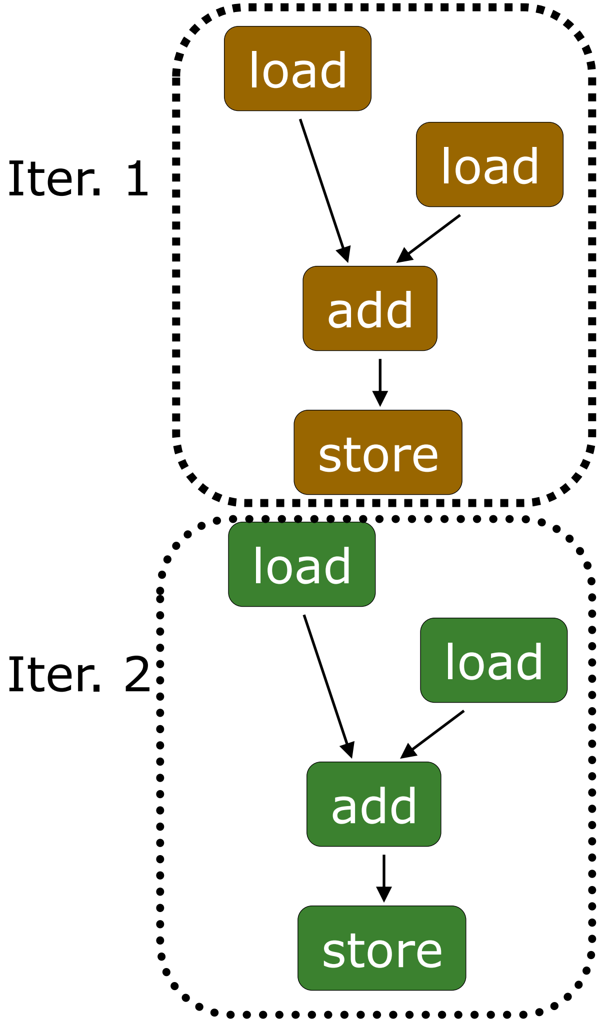
Programming Model vs. Hardware Execution Model

- 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?

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

Scalar Sequential Code



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

1. Sequential (SISD)

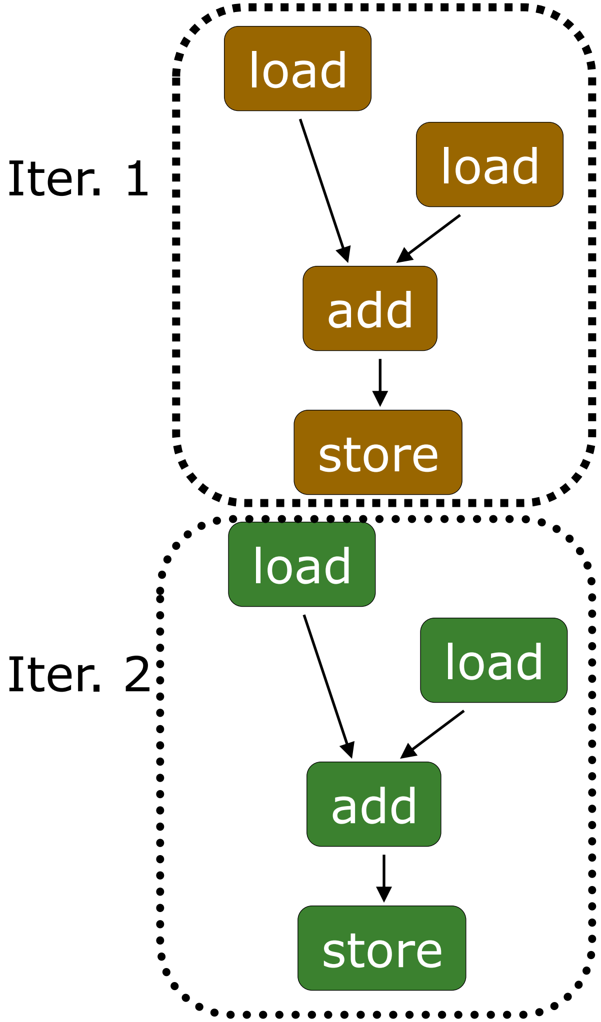
2. Data-Parallel (SIMD)

3. Multithreaded (MIMD/SPMD)

Prog. Model 1: Sequential (SISD)

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

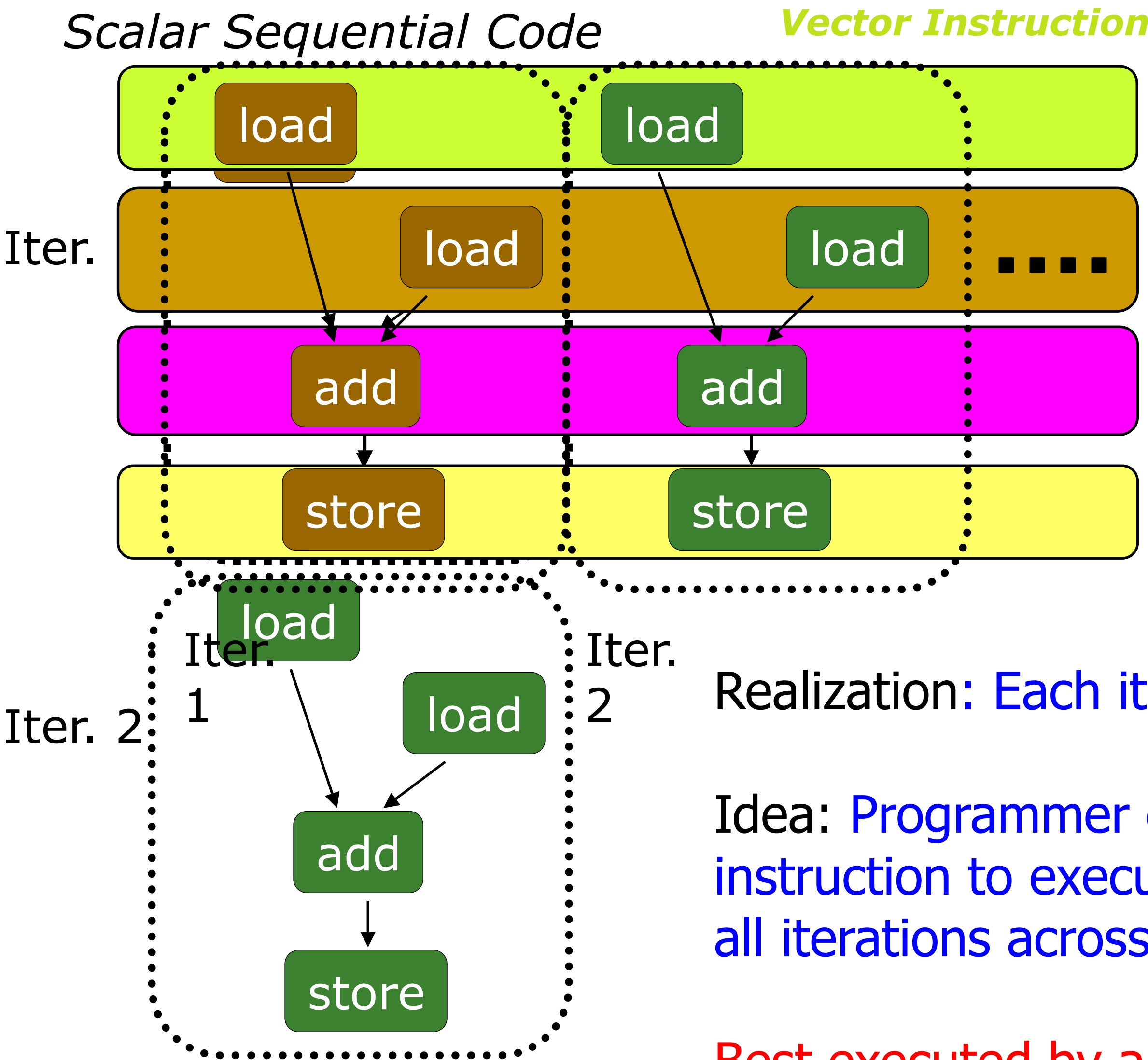
Scalar Sequential Code



- 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];
```



Vectorized Code

VLD A → V1

VLD B → V2

VADD V1 + V2 → V3

VST V3 → C

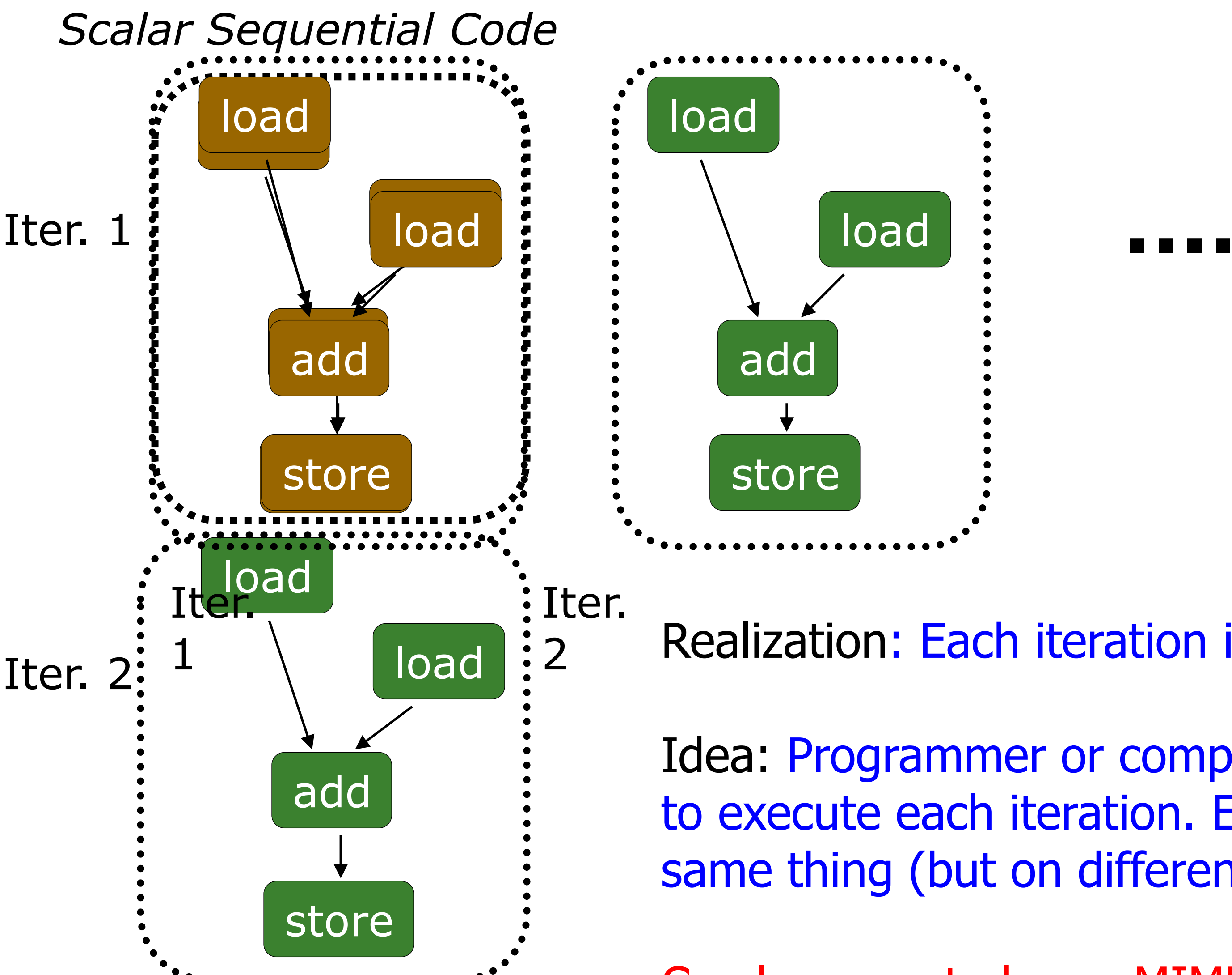
Realization: Each iteration is independent

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

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



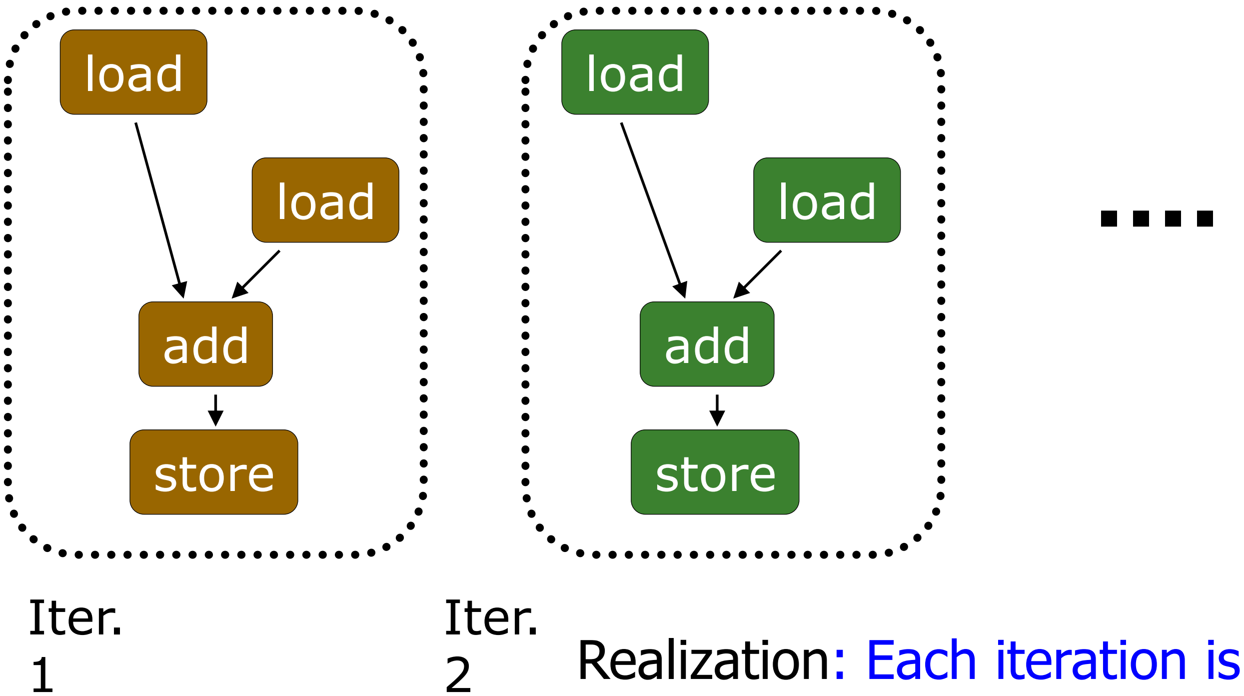
Realization: Each iteration is independent

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

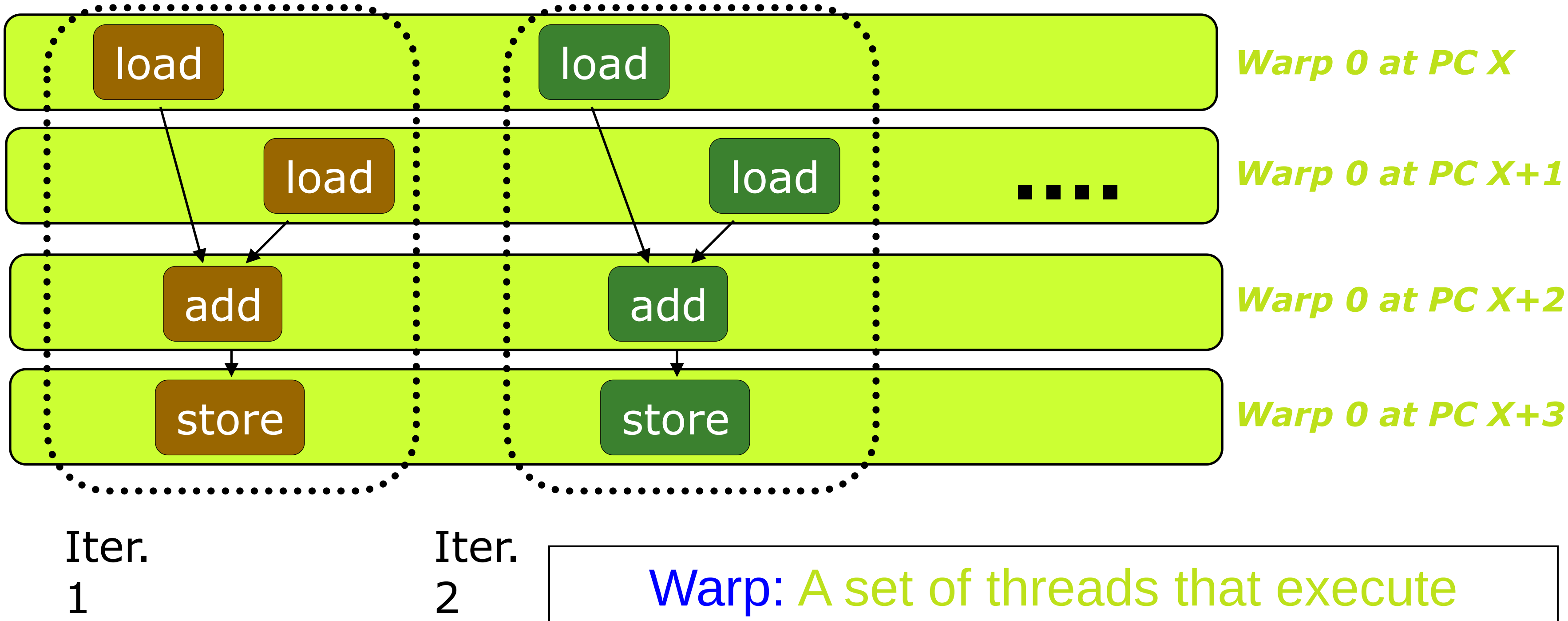
- Except it is **not** programmed using SIMD instructions

- 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];
```



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)

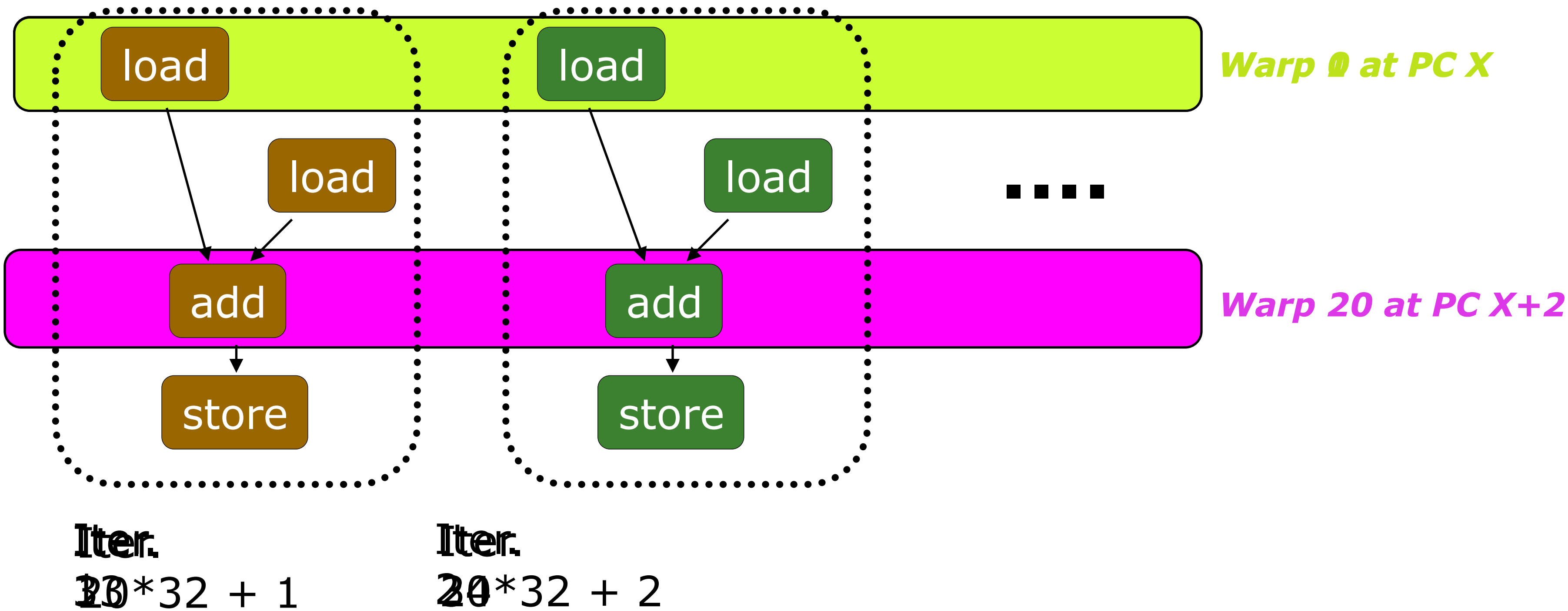
SIMD vs. SIMT Execution Model

- 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

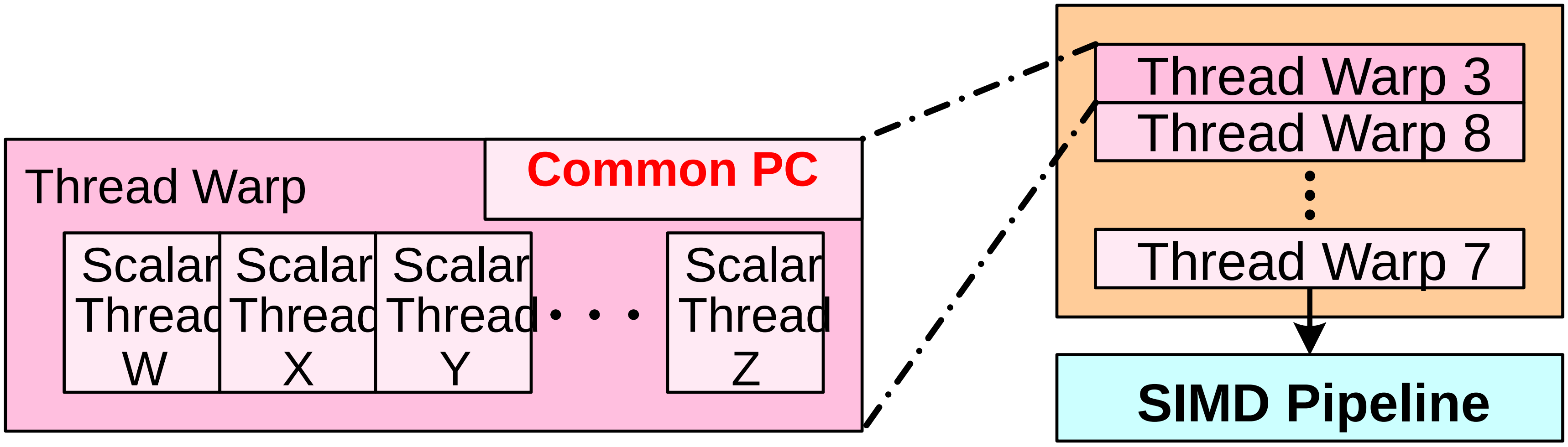
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

- 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



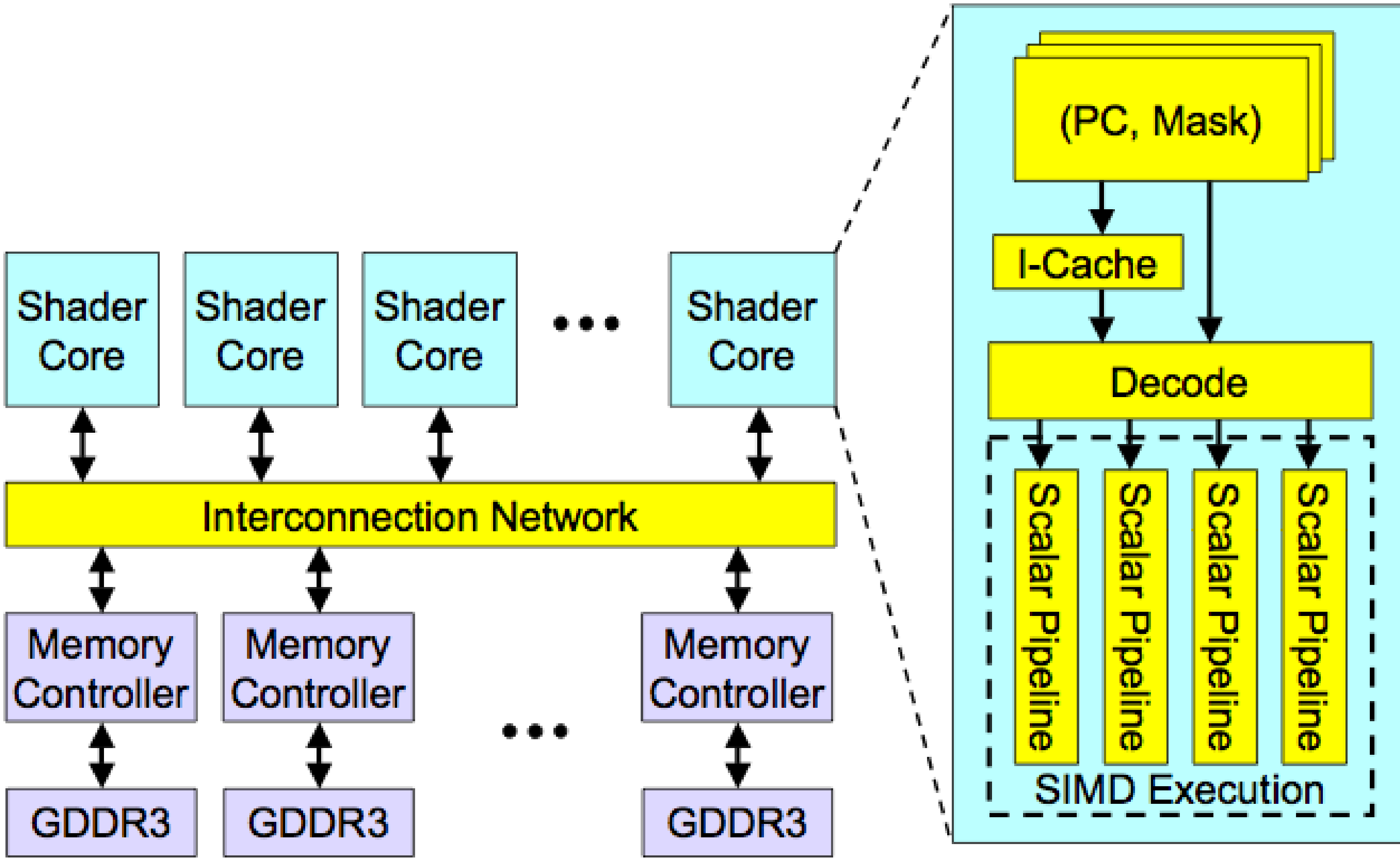
Warps and Warp-Level FGMT

- 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 …



Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.

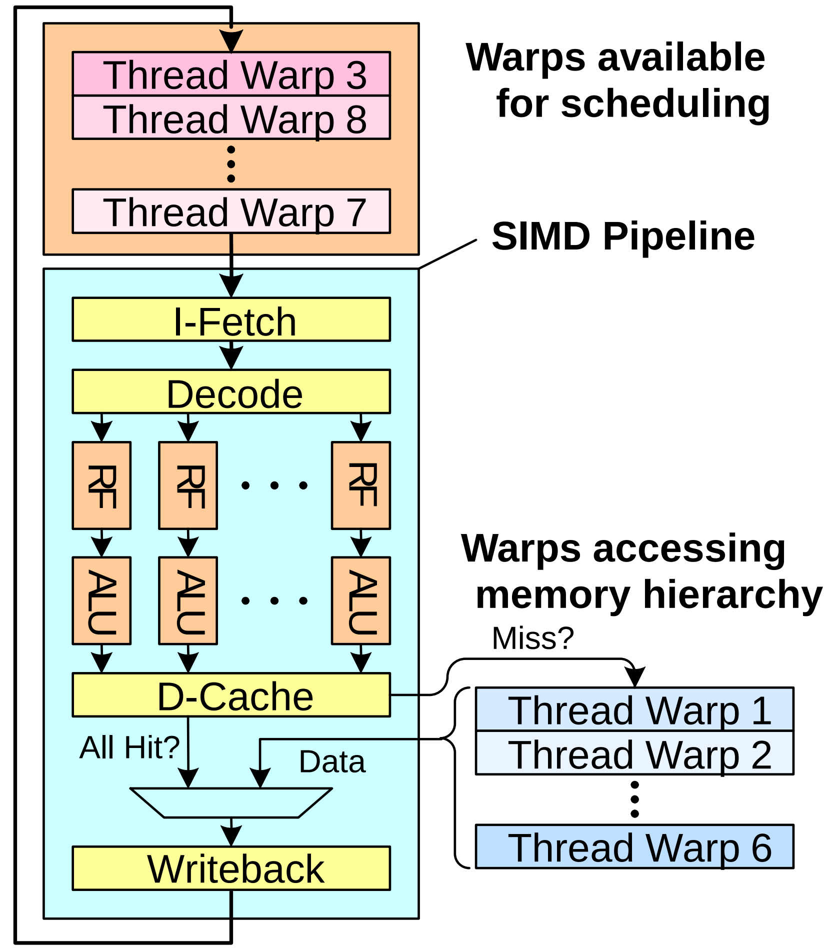
High-Level View of a GPU



Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.

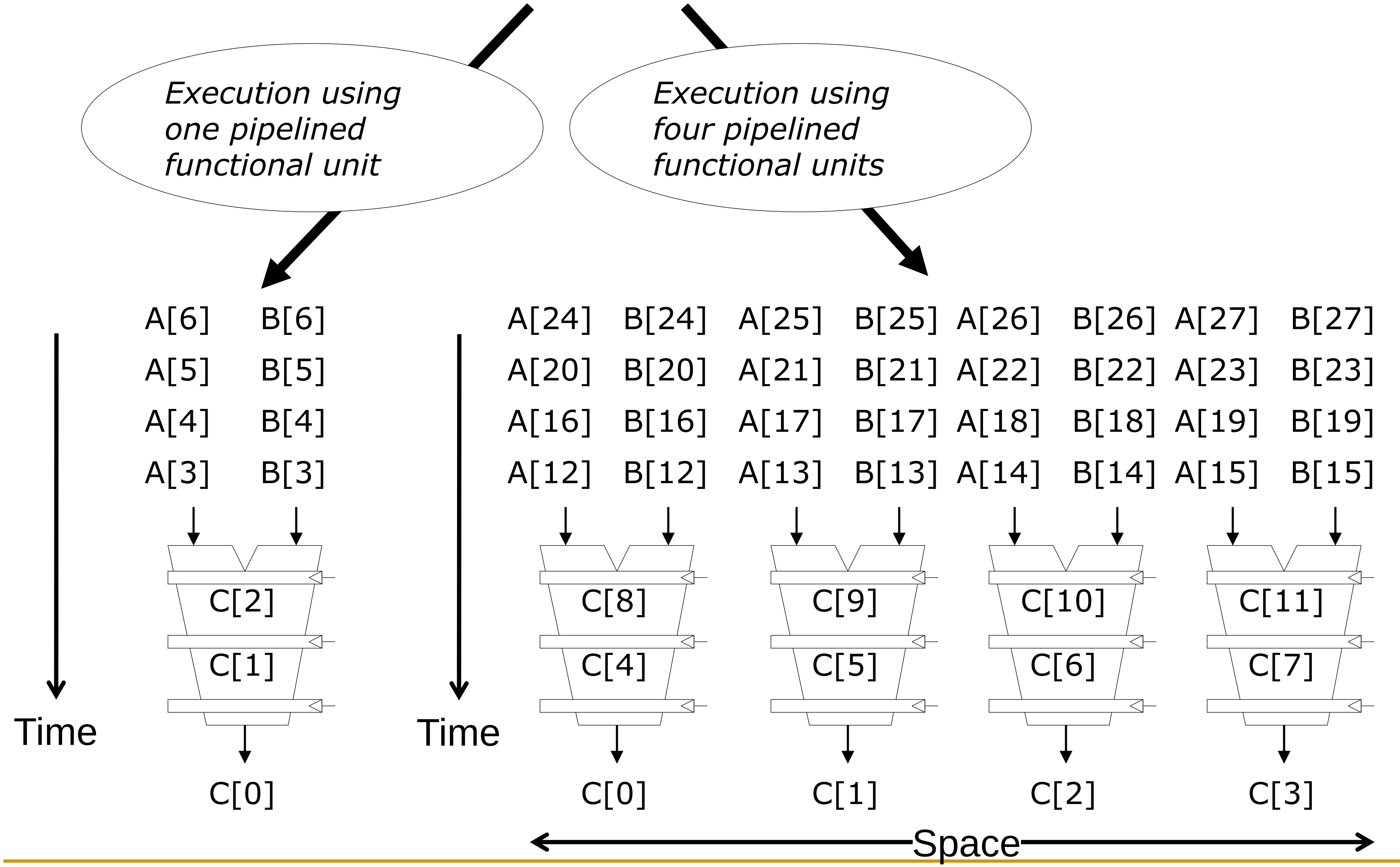
Latency Hiding via Warp-Level FGMT

- 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



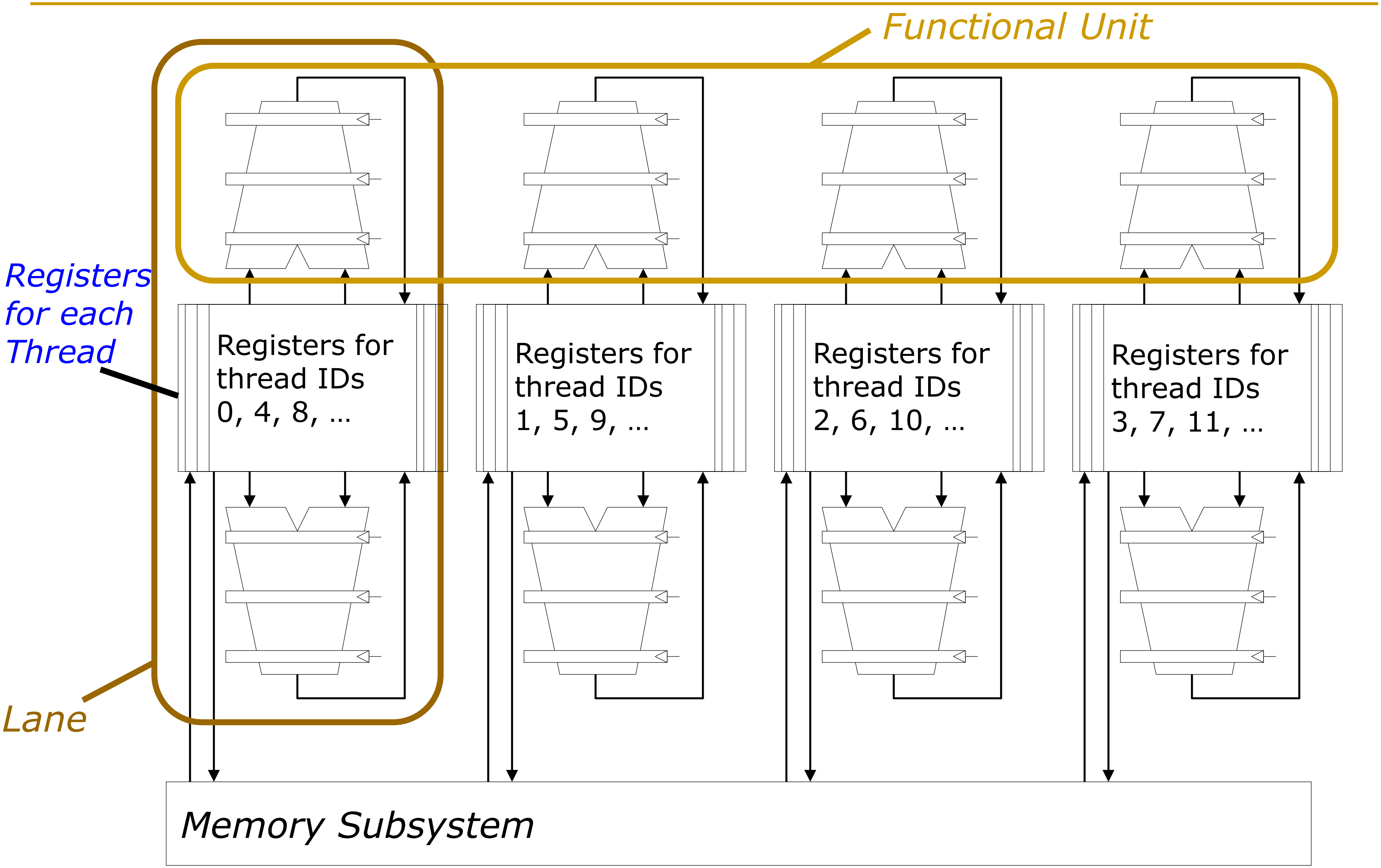
Warp Execution (Recall the Slide)

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



Slide credit: Krste Asanovic

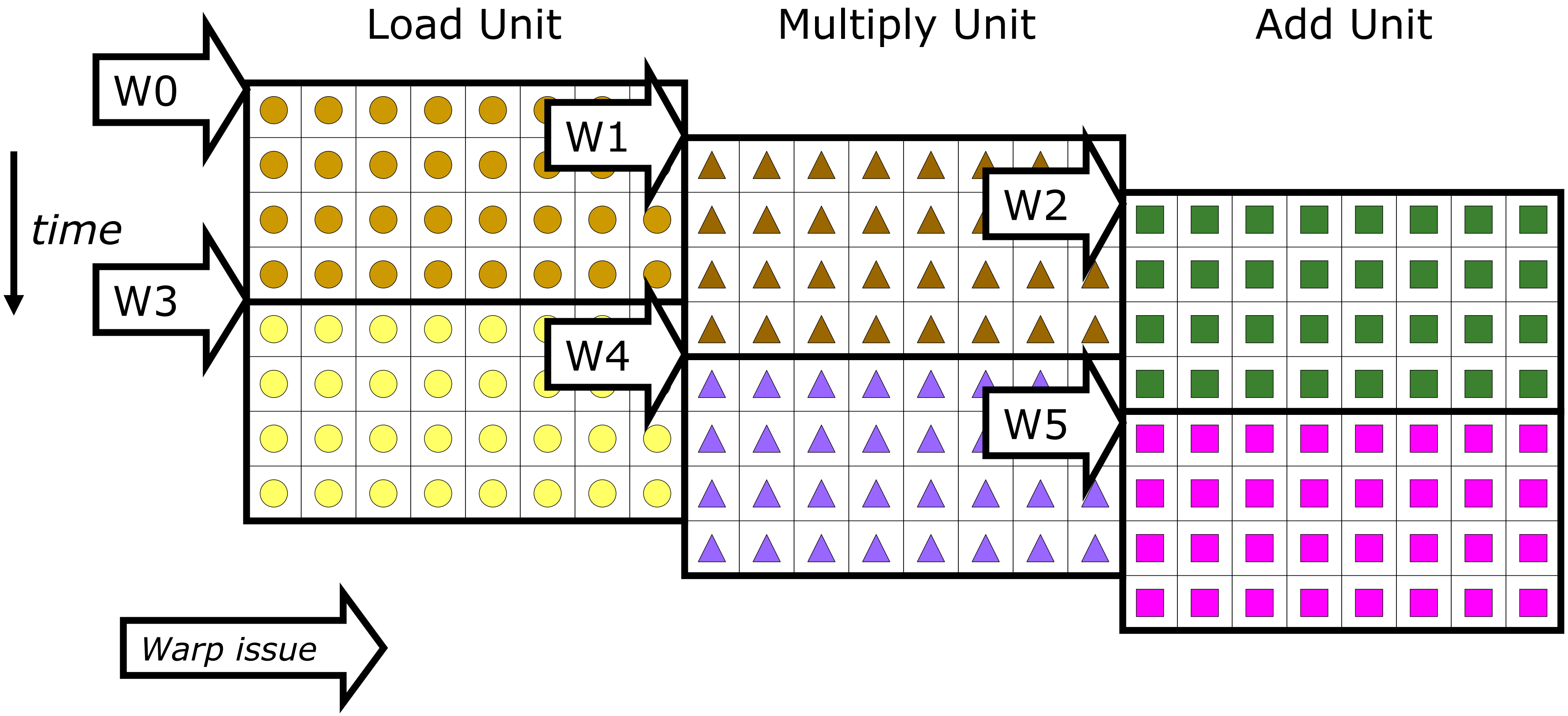
SIMD Execution Unit Structure



Warp Instruction Level Parallelism

Can overlap execution of multiple instructions

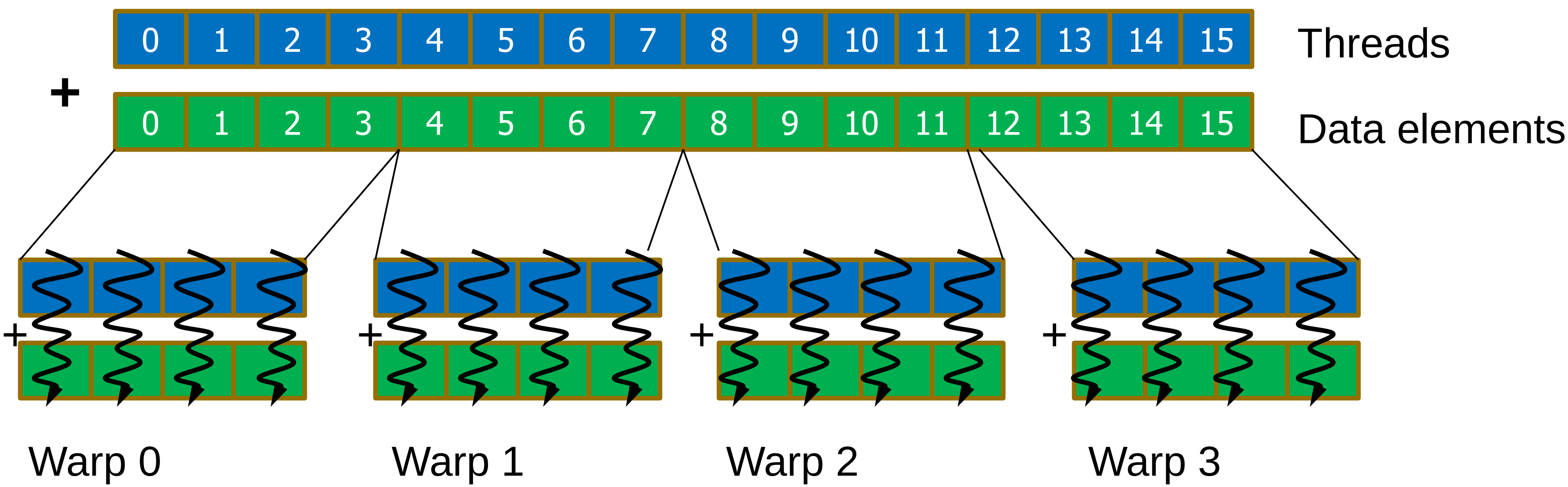
- Example machine has 32 threads per warp and 8 lanes
- Completes 24 operations/cycle while issuing 1 warp/cycle



SIMT Memory Access

- 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



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)

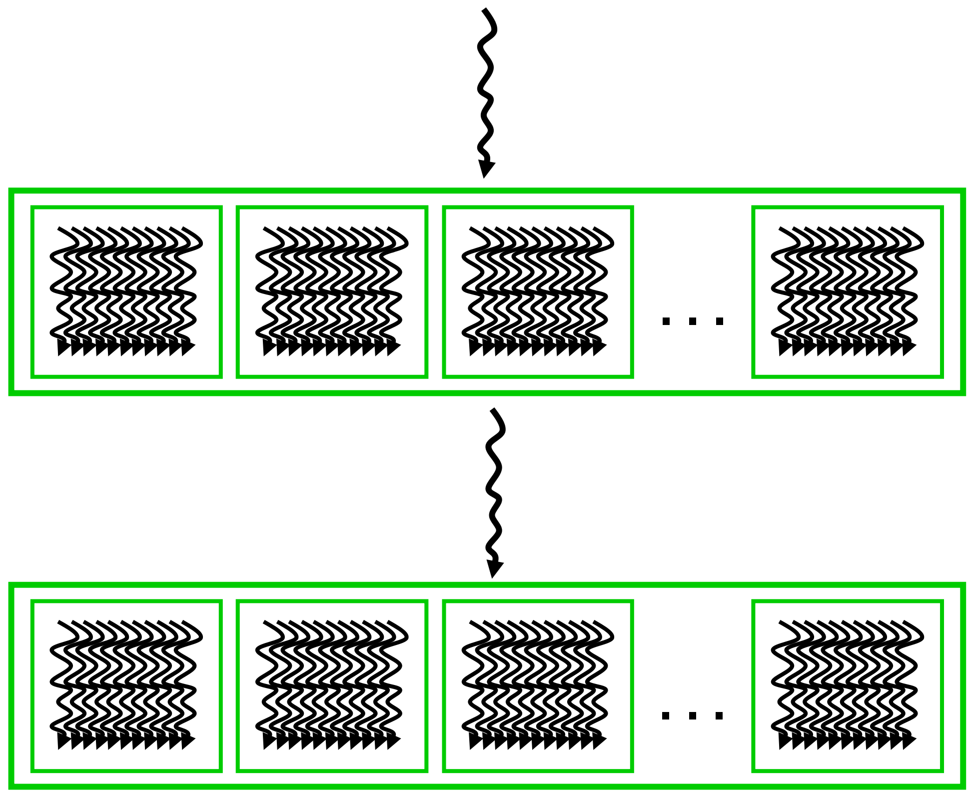
Parallel Kernel (device)

`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) {
C[ii] = A[ii] + B[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

```
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);
}
```

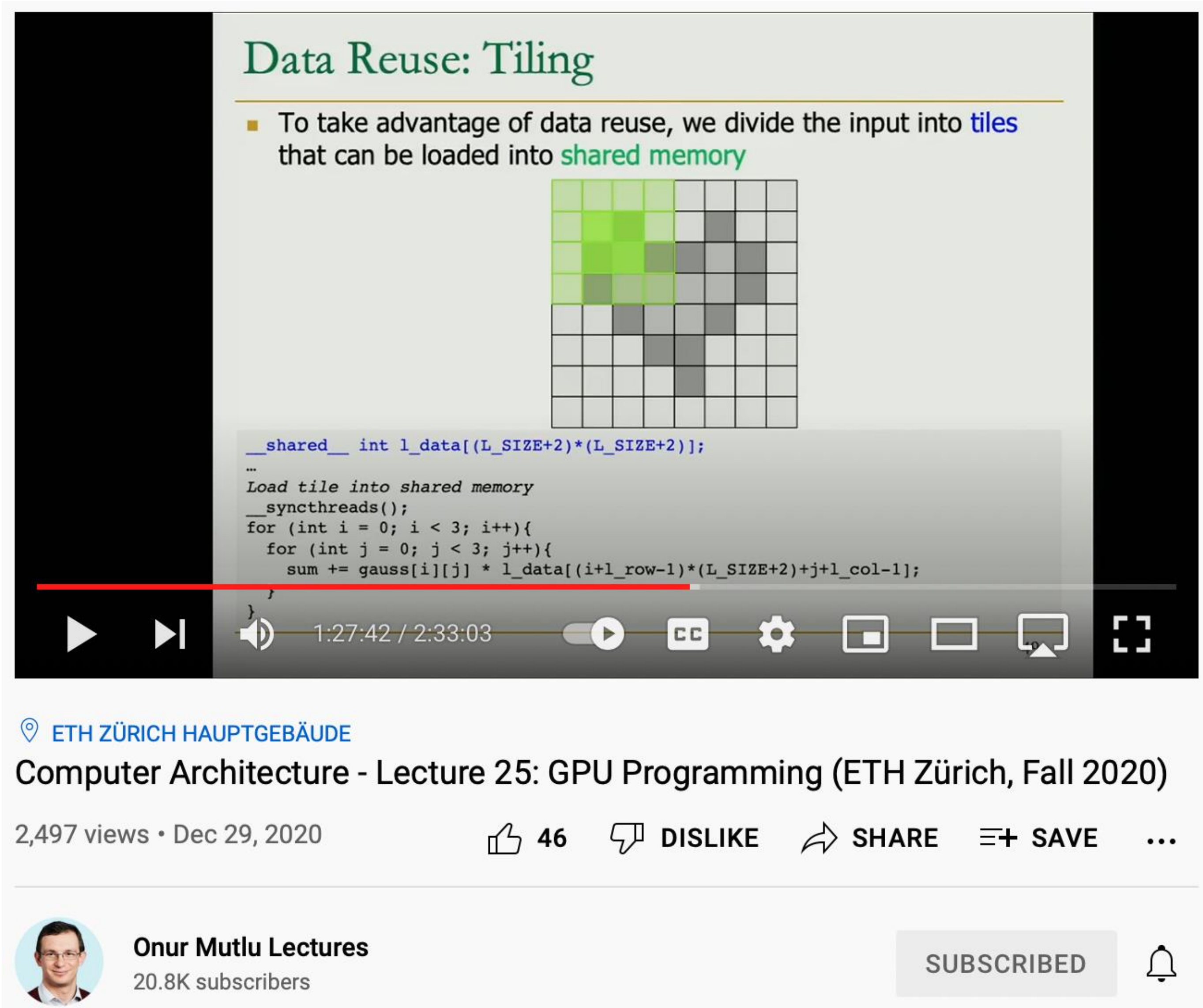
GPU Program

```
__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);
}
```

Slide credit: Hyesoon Kim

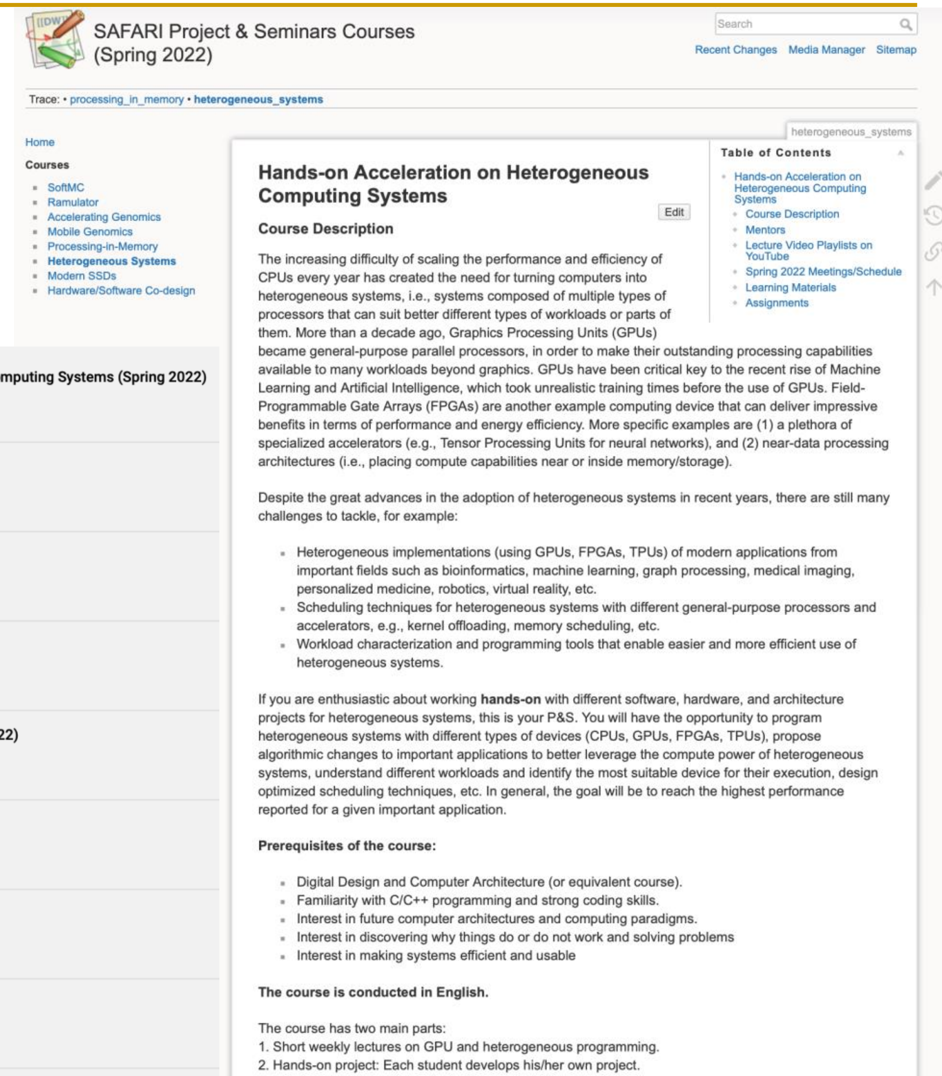
Lecture on GPU Programming



https://youtu.be/AkYnuqVpCug

Heterogeneous Systems Course (Spring 2022)

- Short weekly lectures
- Hands-on projects


SAFARI Project & Seminars Courses (Spring 2022)

Search

Recent Changes Media Manager Sitemap

Trace: • processing_in_memory • **heterogeneous_systems**

heterogeneous_systems

Home

Courses

- SoftMC
- Ramulator
- Accelerating Genomics
- Mobile Genomics
- Processing-in-Memory
- **Heterogeneous Systems**
- Modern SSDs
- Hardware/Software Co-design

Table of Contents

- Hands-on Acceleration on Heterogeneous Computing Systems
  - Course Description
  - Mentors
  - Lecture Video Playlists on YouTube
  - Spring 2022 Meetings/Schedule
  - Learning Materials
  - Assignments

Hands-on Acceleration on Heterogeneous Computing Systems

Edit

Course Description

The increasing difficulty of scaling the performance and efficiency of CPUs every year has created the need for turning computers into heterogeneous systems, i.e., systems composed of multiple types of processors that can suit better different types of workloads or parts of them. More than a decade ago, Graphics Processing Units (GPUs) became general-purpose parallel processors, in order to make their outstanding processing capabilities available to many workloads beyond graphics. GPUs have been critical key to the recent rise of Machine Learning and Artificial Intelligence, which took unrealistic training times before the use of GPUs. Field-Programmable Gate Arrays (FPGAs) are another example computing device that can deliver impressive benefits in terms of performance and energy efficiency. More specific examples are (1) a plethora of specialized accelerators (e.g., Tensor Processing Units for neural networks), and (2) near-data processing architectures (i.e., placing compute capabilities near or inside memory/storage).

Despite the great advances in the adoption of heterogeneous systems in recent years, there are still many challenges to tackle, for example:

- Heterogeneous implementations (using GPUs, FPGAs, TPUs) of modern applications from important fields such as bioinformatics, machine learning, graph processing, medical imaging, personalized medicine, robotics, virtual reality, etc.
- Scheduling techniques for heterogeneous systems with different general-purpose processors and accelerators, e.g., kernel offloading, memory scheduling, etc.
- Workload characterization and programming tools that enable easier and more efficient use of heterogeneous systems.

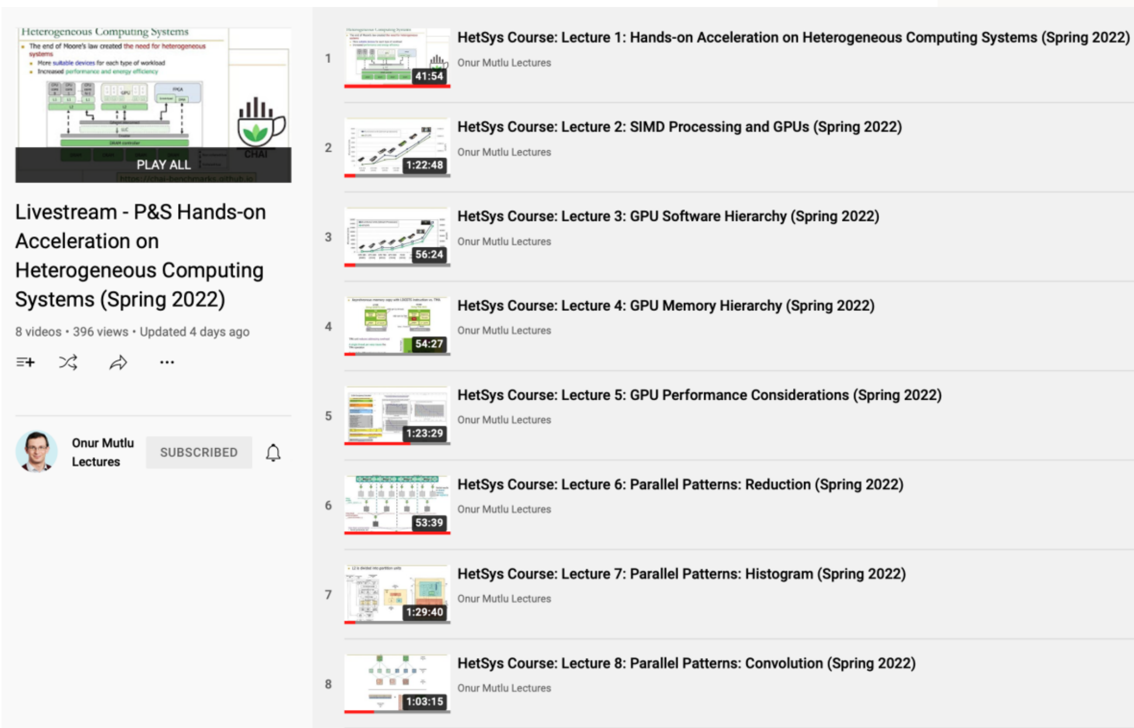
If you are enthusiastic about working **hands-on** with different software, hardware, and architecture projects for heterogeneous systems, this is your P&S. You will have the opportunity to program heterogeneous systems with different types of devices (CPUs, GPUs, FPGAs, TPUs), propose algorithmic changes to important applications to better leverage the compute power of heterogeneous systems, understand different workloads and identify the most suitable device for their execution, design optimized scheduling techniques, etc. In general, the goal will be to reach the highest performance reported for a given important application.

Prerequisites of the course:

- Digital Design and Computer Architecture (or equivalent course).
- Familiarity with C/C++ programming and strong coding skills.
- Interest in future computer architectures and computing paradigms.
- Interest in discovering why things do or do not work and solving problems
- Interest in making systems efficient and usable

The course is conducted in English.

The course has two main parts:
1. Short weekly lectures on GPU and heterogeneous programming.
2. Hands-on project: Each student develops his/her own project.

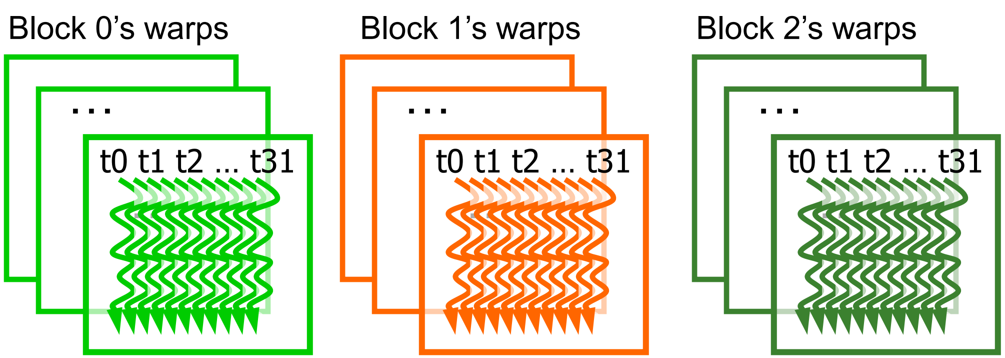


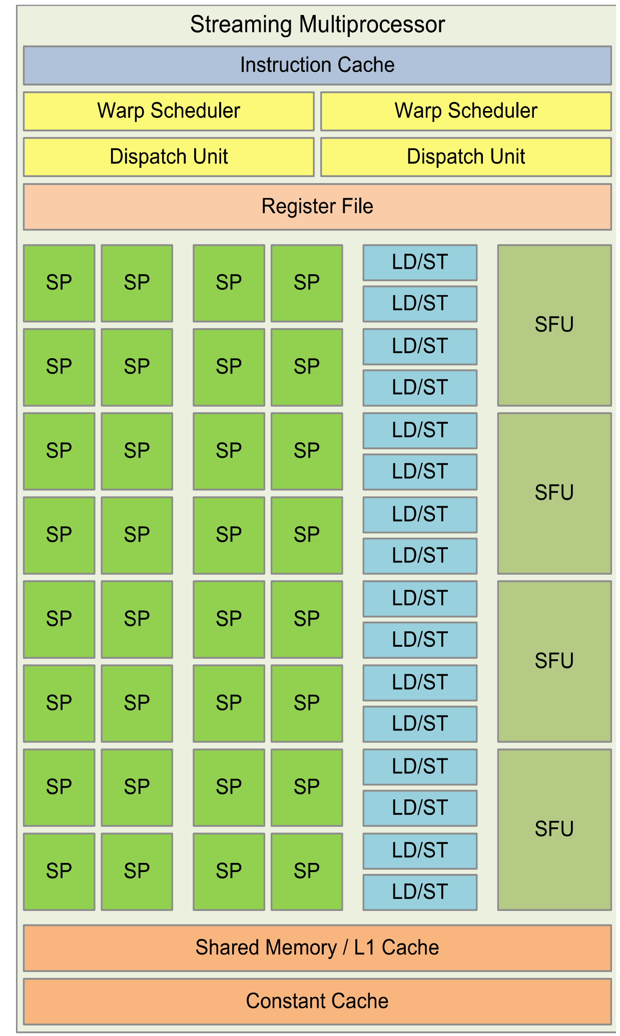
https://youtube.com/playlist?list=PL5Q2soXY2Zi9XrgXR38IM_FTjmY6h7Gzm

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

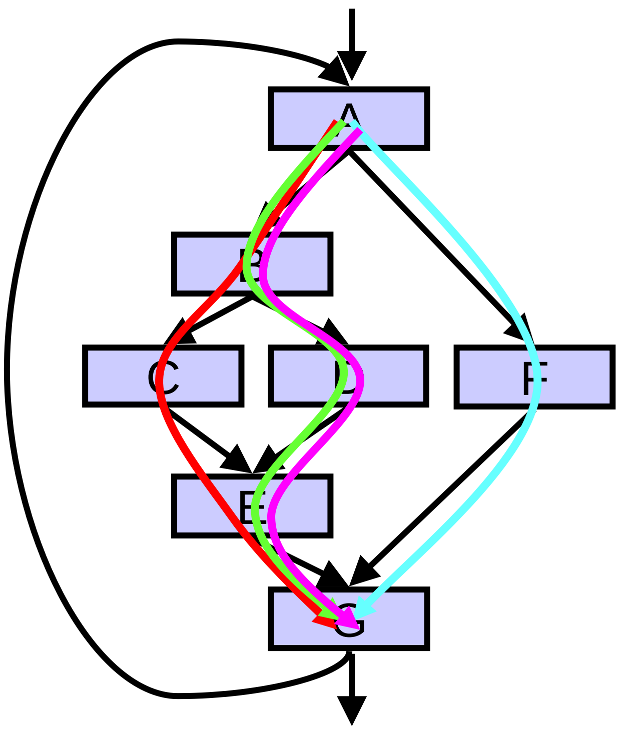
- SIMD: A single sequential instruction stream of SIMD instructions → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN

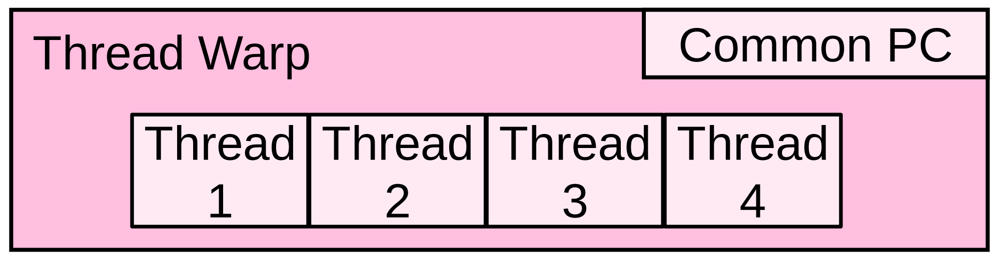
- SIMT: Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

- 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



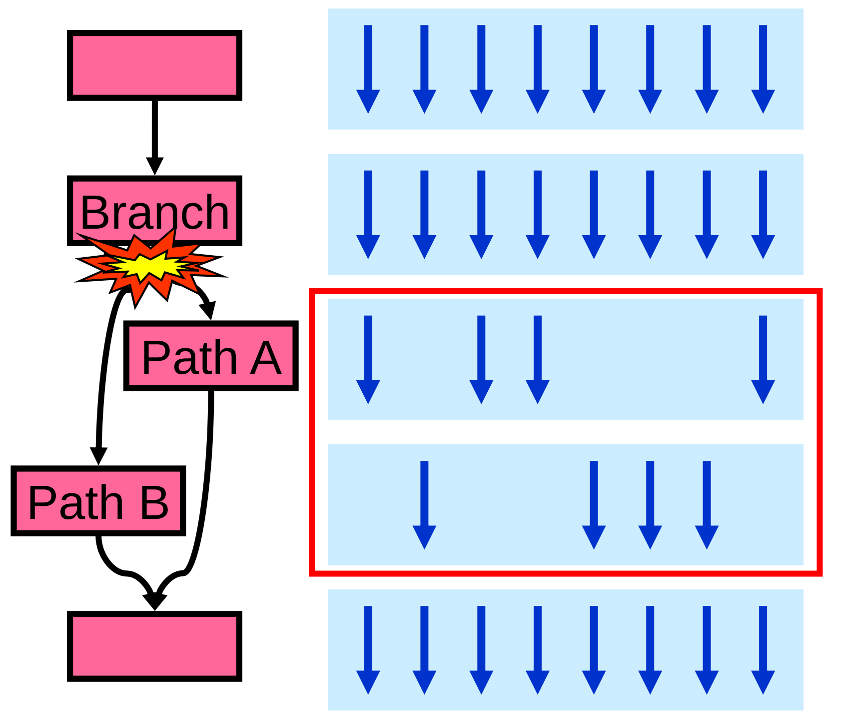


Slide credit: Tor Aamodt

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?

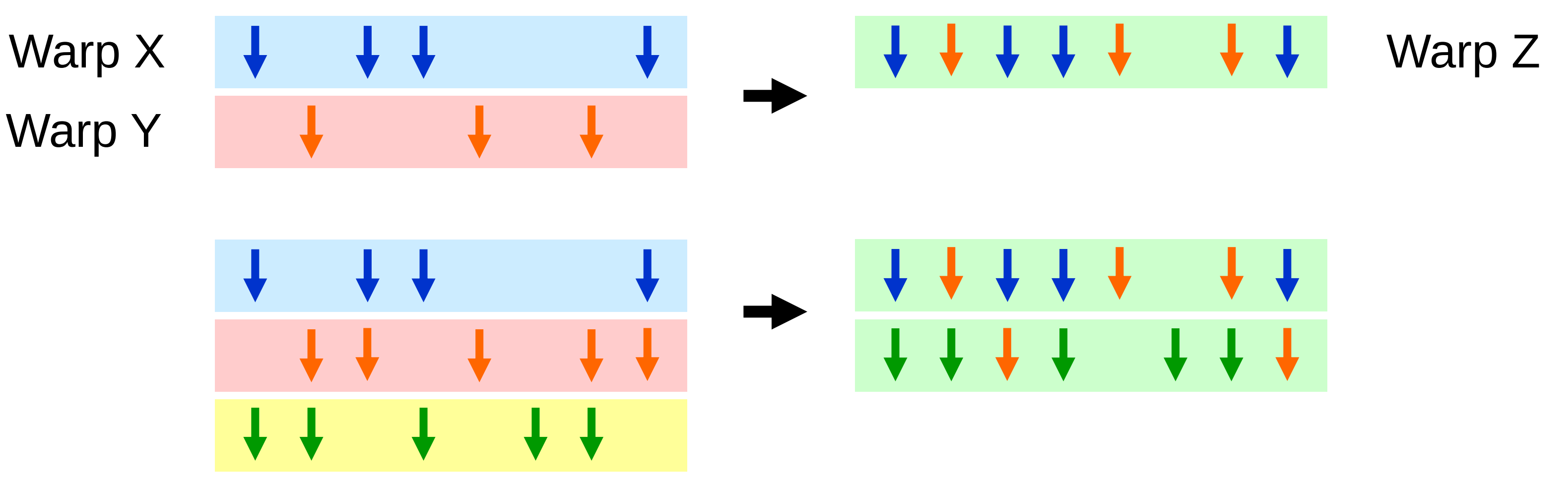
Remember: Each Thread Is Independent

- 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)

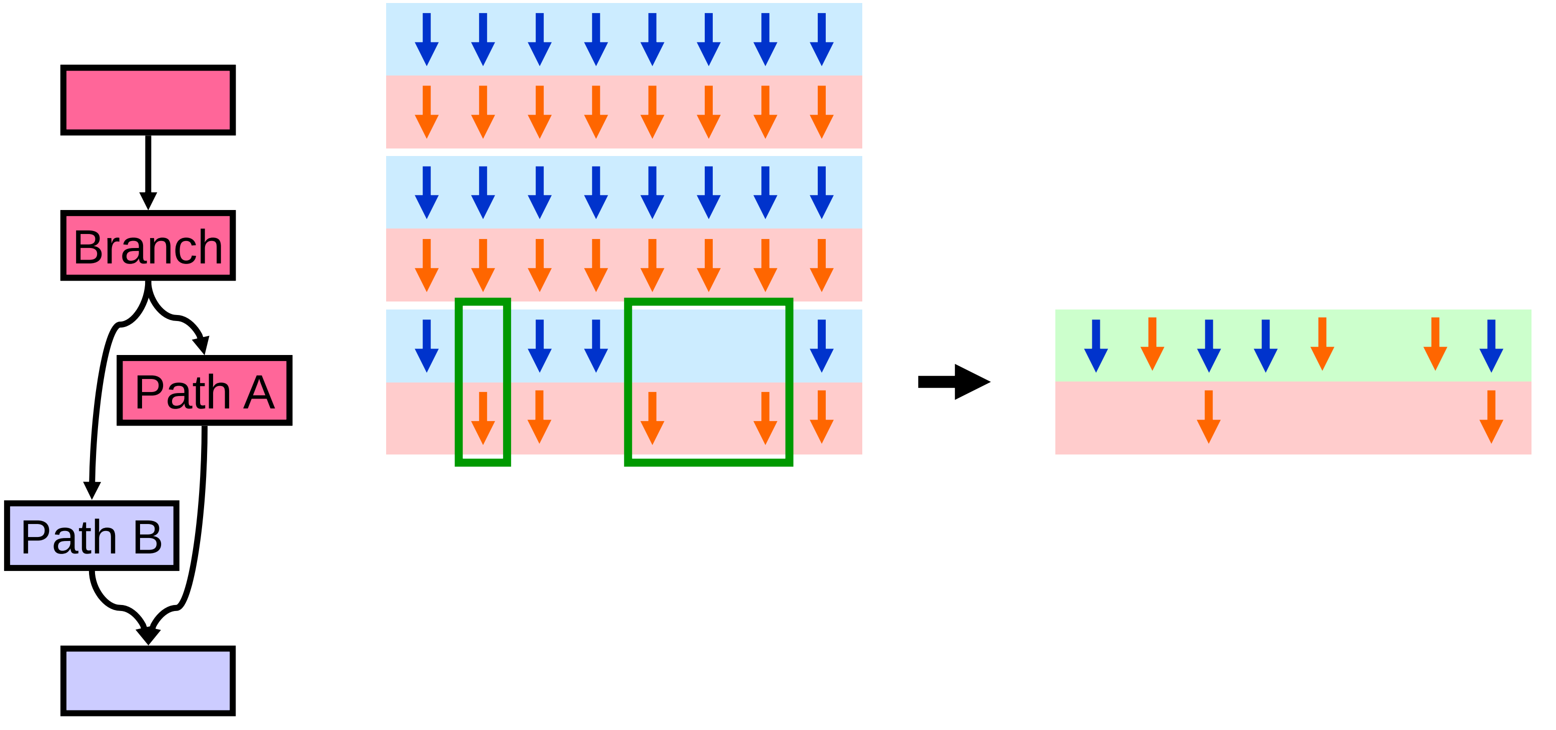
Dynamic Warp Formation/Merging

- 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



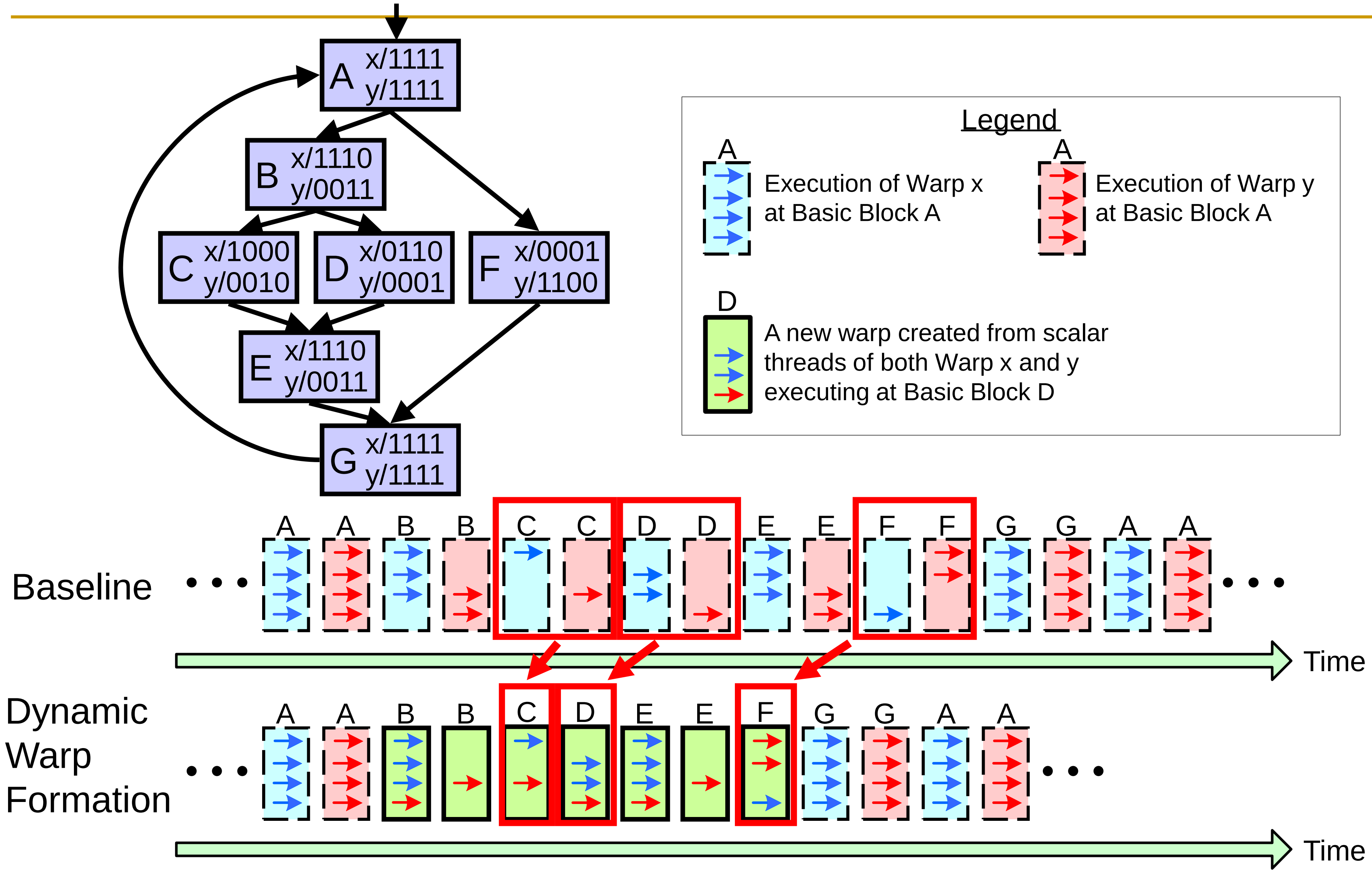
Dynamic Warp Formation/Merging

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



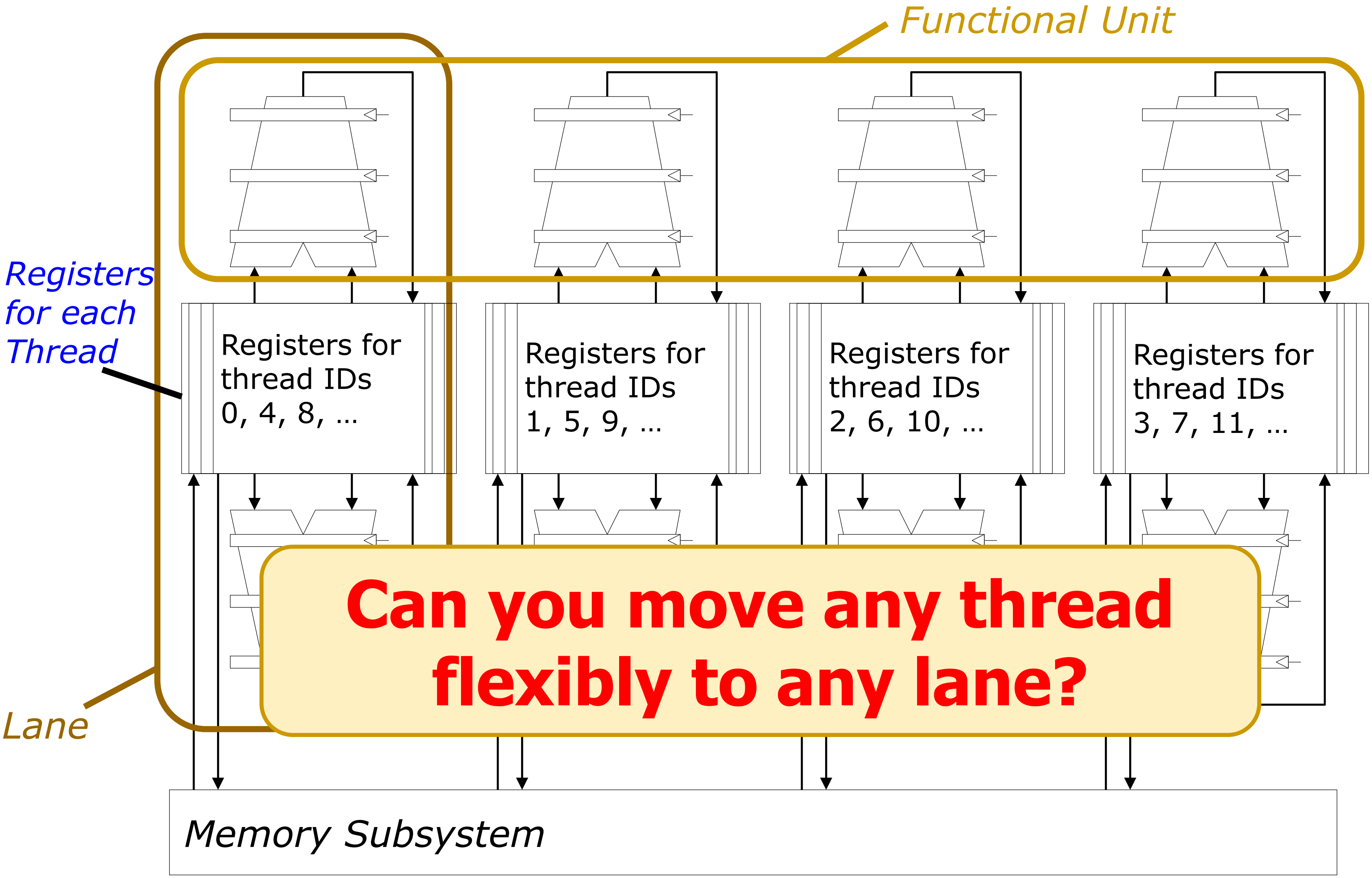
- Fung et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," MICRO 2007.

Dynamic Warp Formation Example



Slide credit: Tor Aamodt

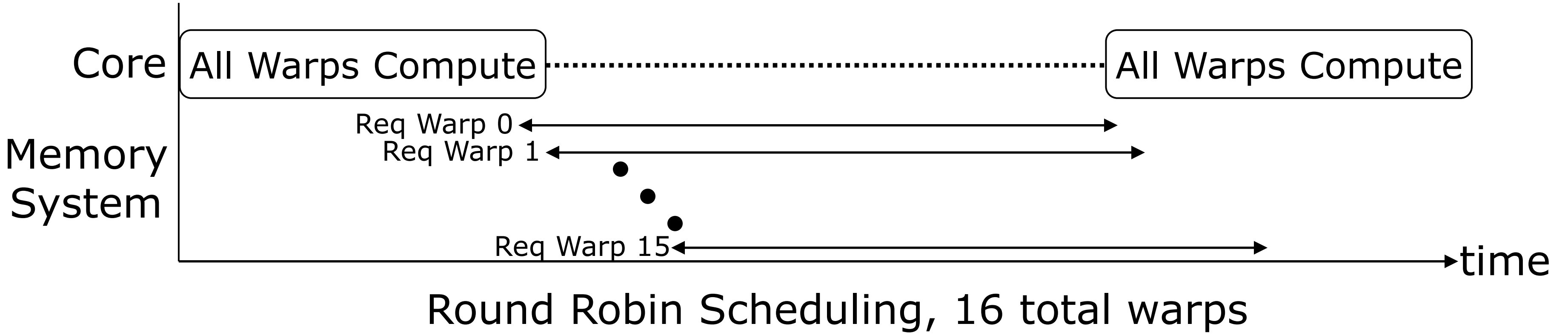
Hardware Constraints Limit Flexibility of Warp Grouping



Slide credit: Krste Asanovic

Large Warps and Two-Level Warp Scheduling

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

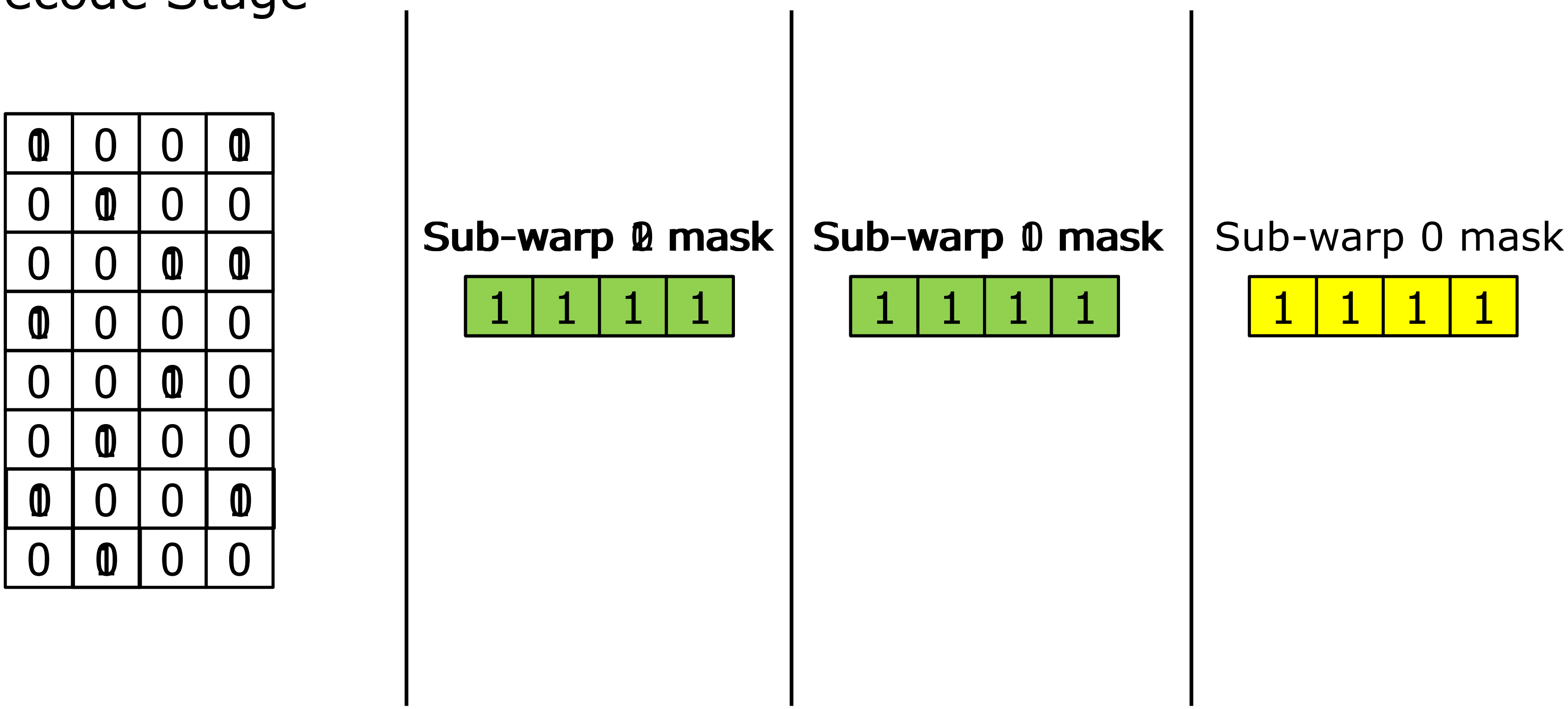


Narasiman et al., "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," MICRO 2011.

Large Warp Microarchitecture Example

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

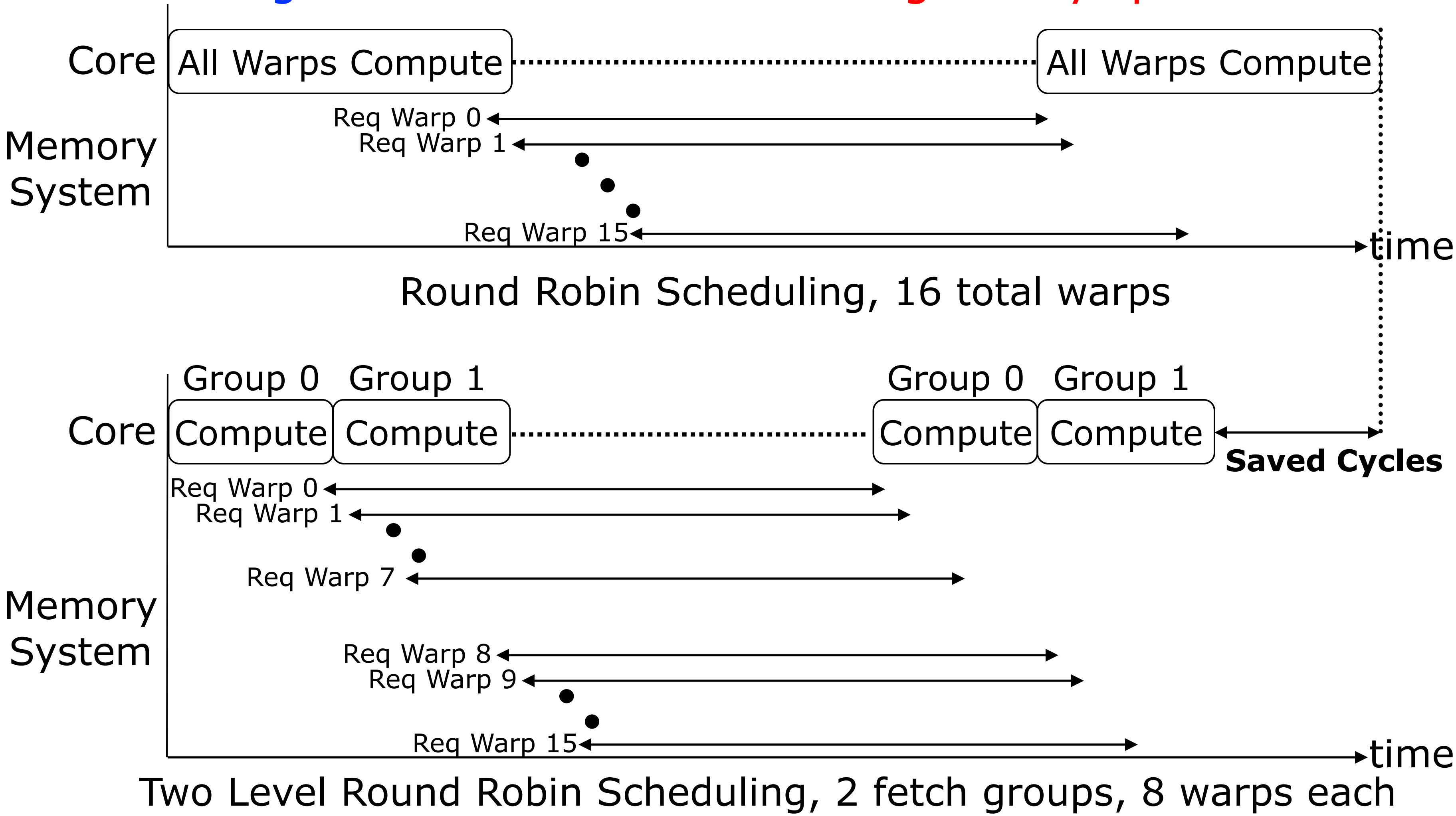
Decode Stage



Narasiman et al., "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," MICRO 2011.

Two-Level Round Robin

- Scheduling in two levels to deal with long latency operations



Narasiman et al., "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," MICRO 2011.

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)
A previous version as HPS Technical Report, TR-HPS-2010-006, December 2010.

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

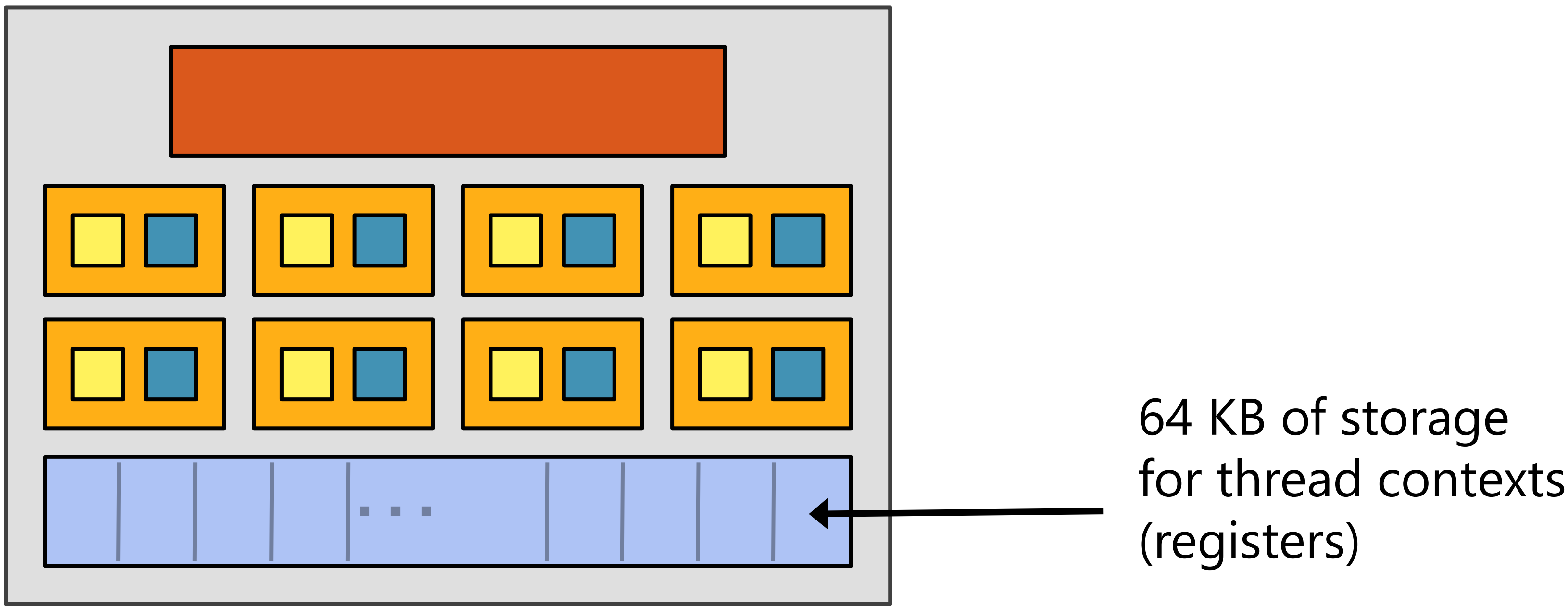
NVIDIA GeForce GTX 285

- NVIDIA-speak:
  - 240 stream processors
  - "SIMT execution"

- Generic speak:
  - 30 cores
  - 8 SIMD functional units per core

- NVIDIA, "NVIDIA GeForce GTX 200 GPU. Architectural Overview. White Paper," 2008.

NVIDIA GeForce GTX 285 “core”



= SIMD functional unit, control shared across 8 units

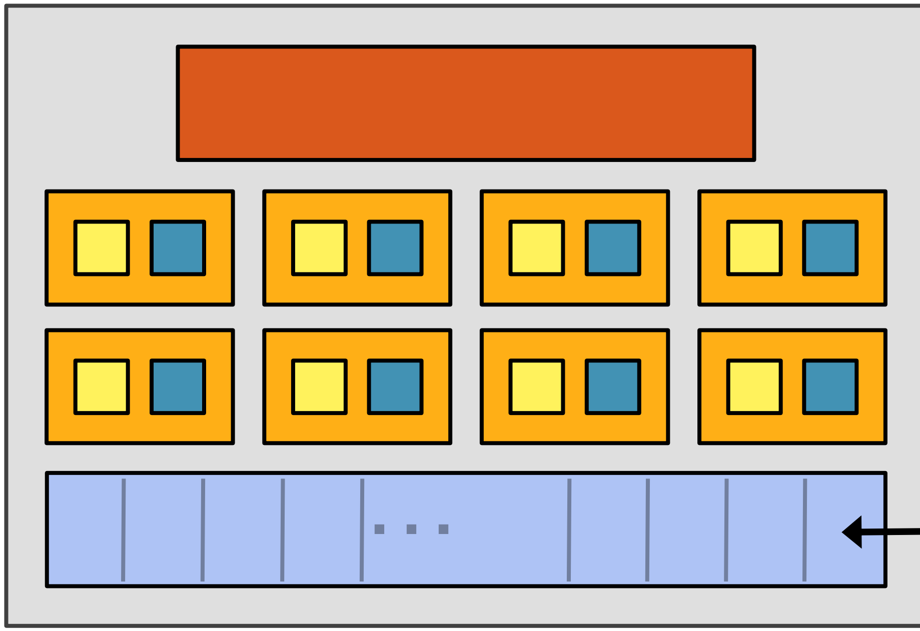
= multiply-add

= multiply

= instruction stream decode

= execution context storage

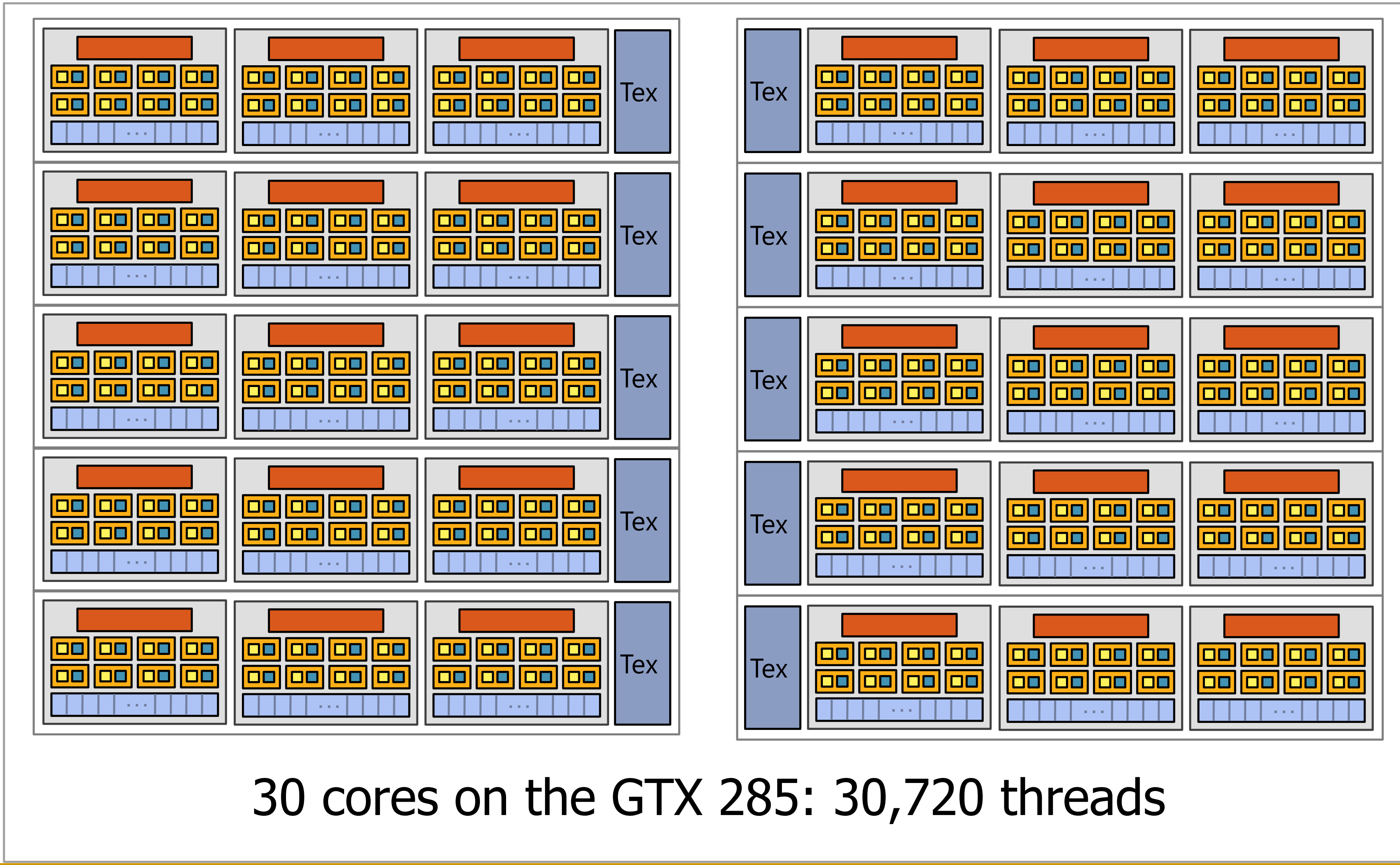
NVIDIA GeForce GTX 285 “core”



64 KB of storage
for thread contexts
(registers)

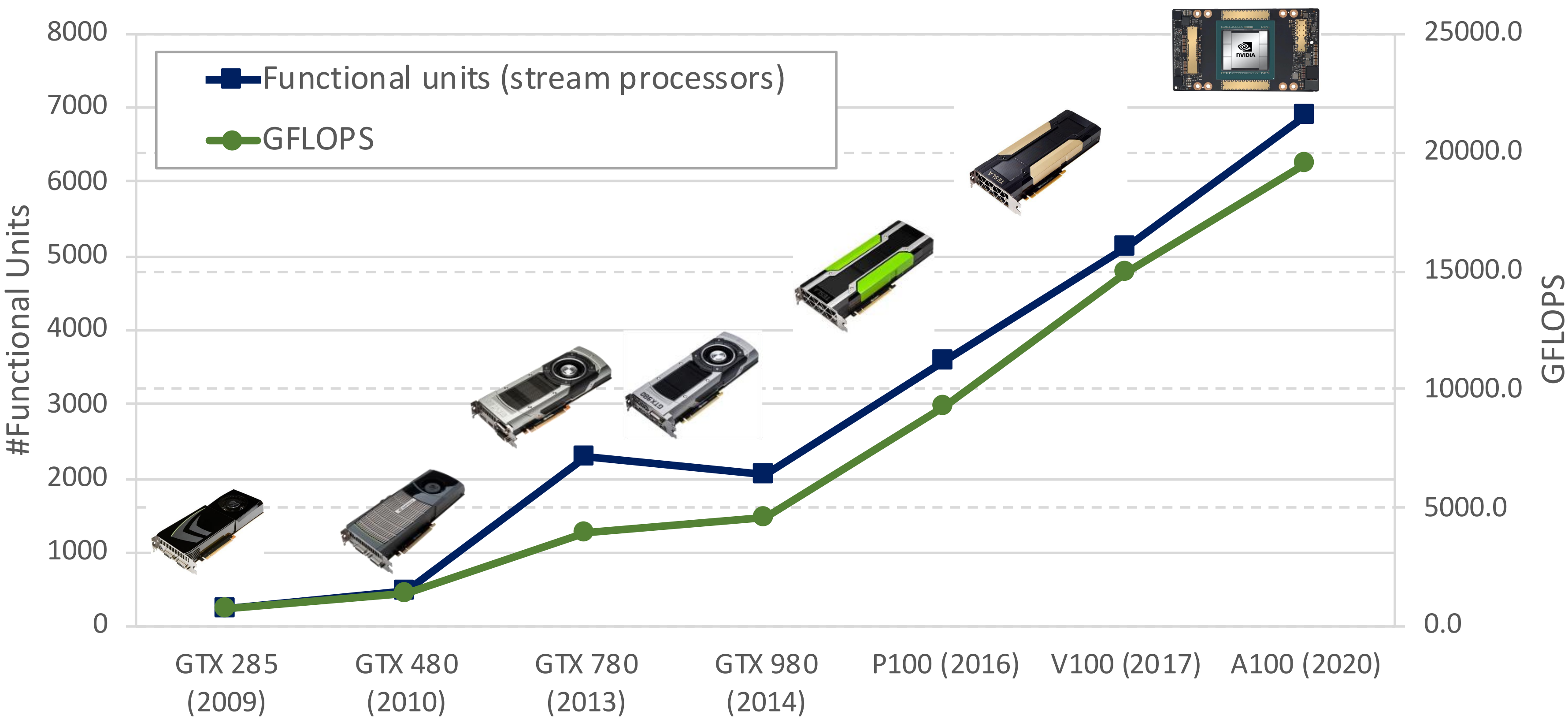
- Groups of 32 threads share instruction stream (each group is a Warp)
- Up to 32 warps are simultaneously interleaved
- Up to 1024 thread contexts can be stored

NVIDIA GeForce GTX 285



Slide credit: Kayvon Fatahalian

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

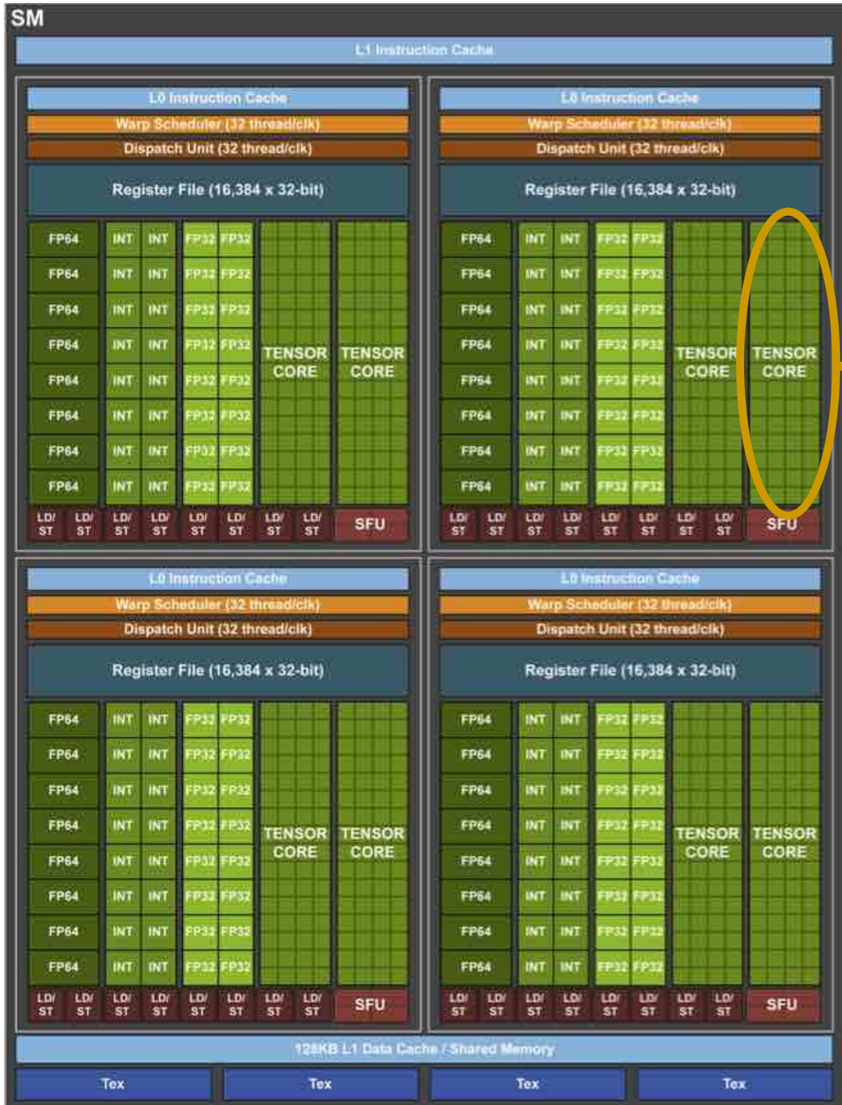
- NVIDIA, "NVIDIA Tesla V100 GPU Architecture. White Paper," 2017.

NVIDIA V100 Block Diagram

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

80 cores on the V100

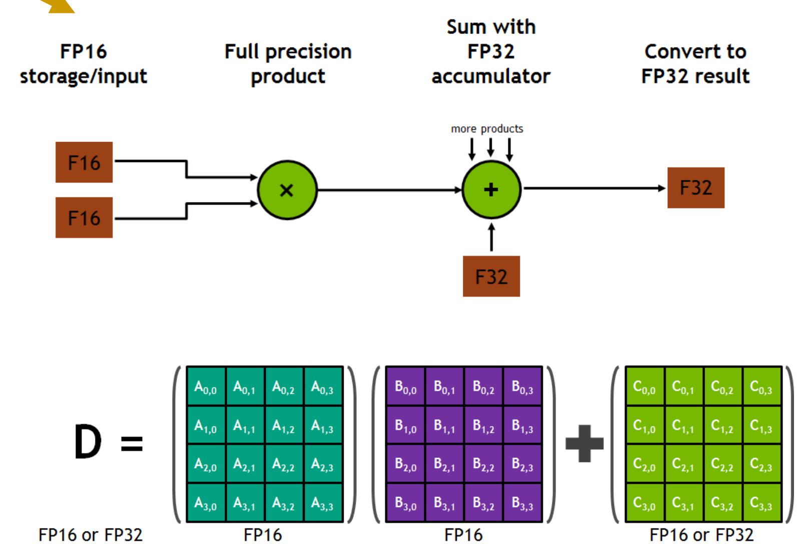
NVIDIA V100 Core



15.7 TFLOPS Single Precision

7.8 TFLOPS Double Precision

125 TFLOPS for Deep Learning (Tensor cores)

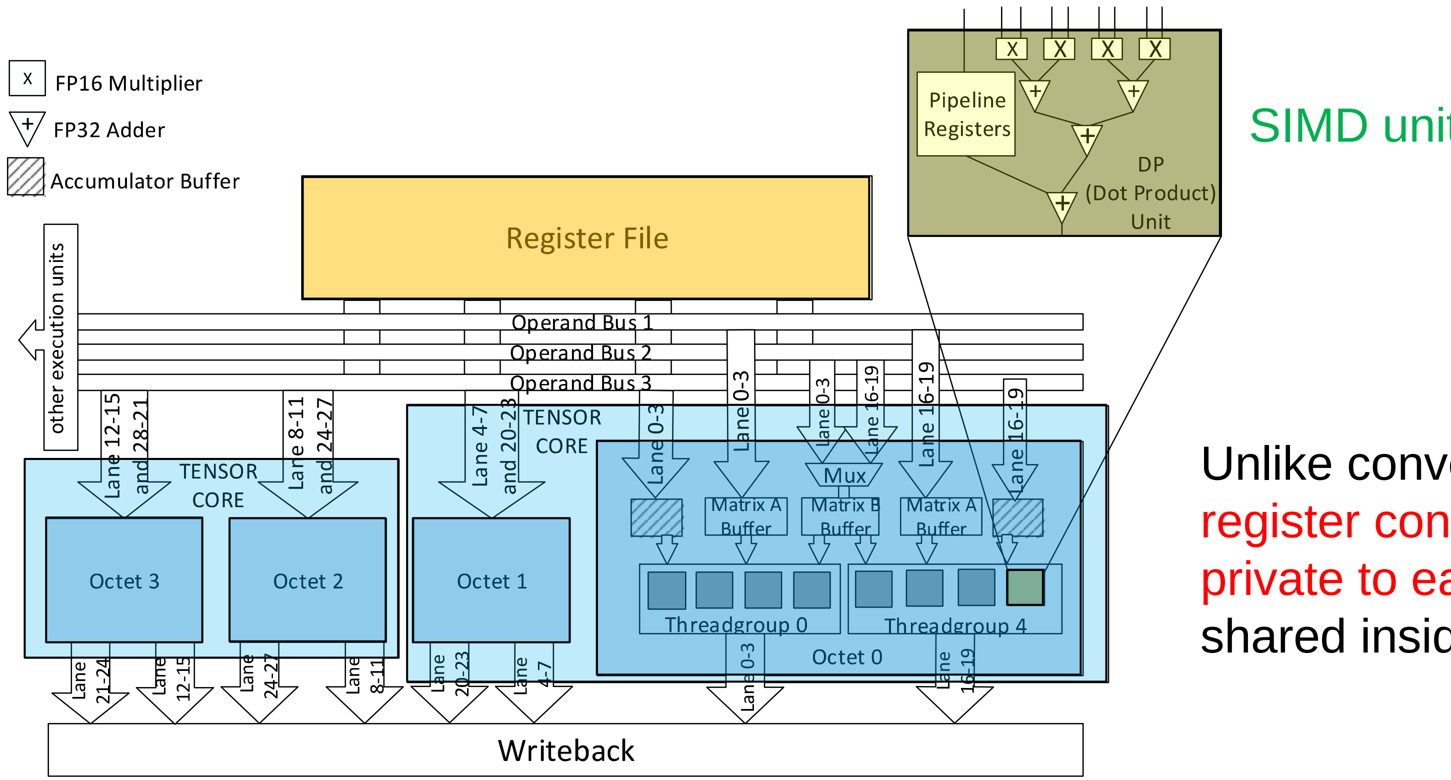


$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

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

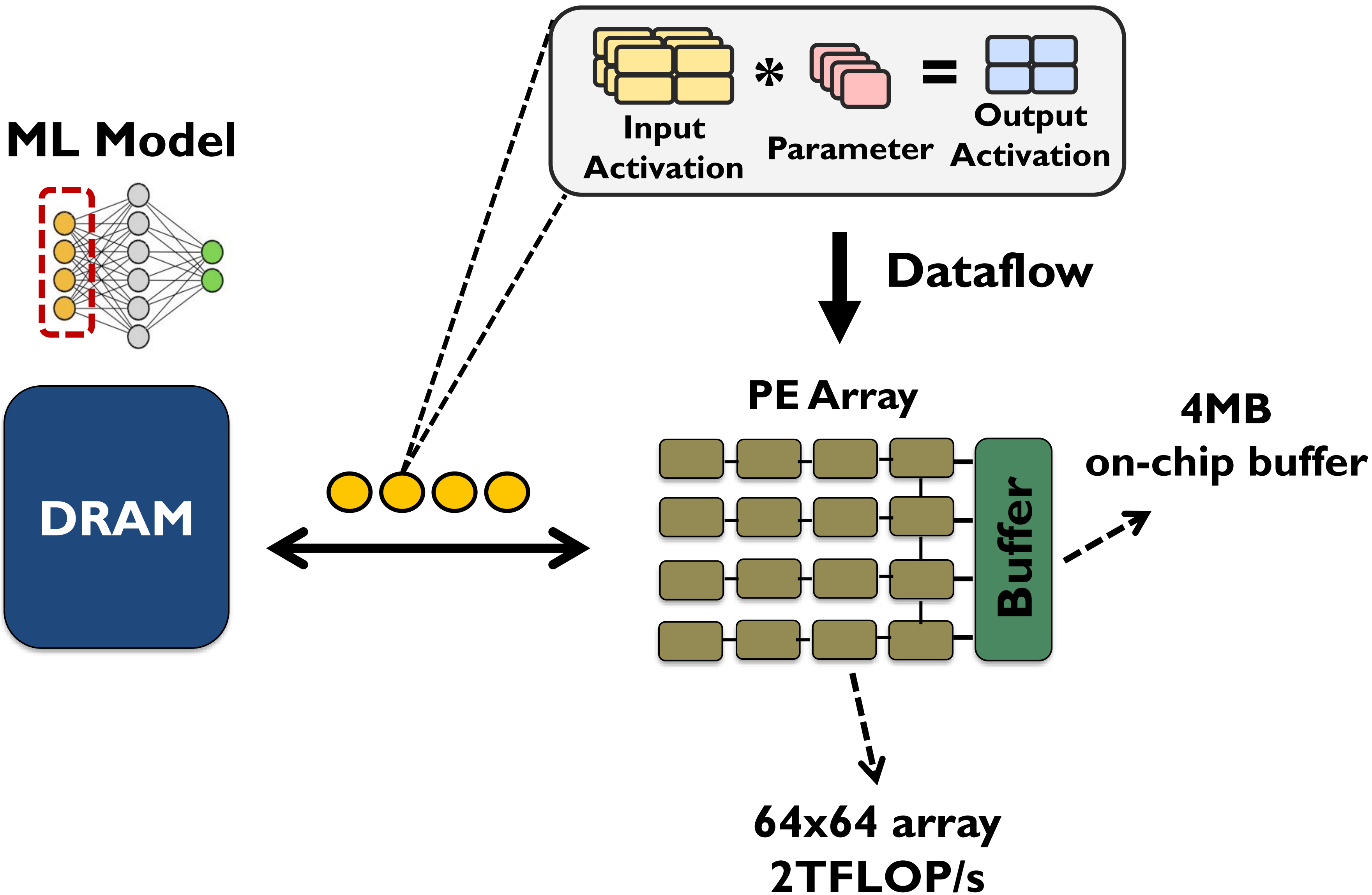


Proposed* tensor core microarchitecture

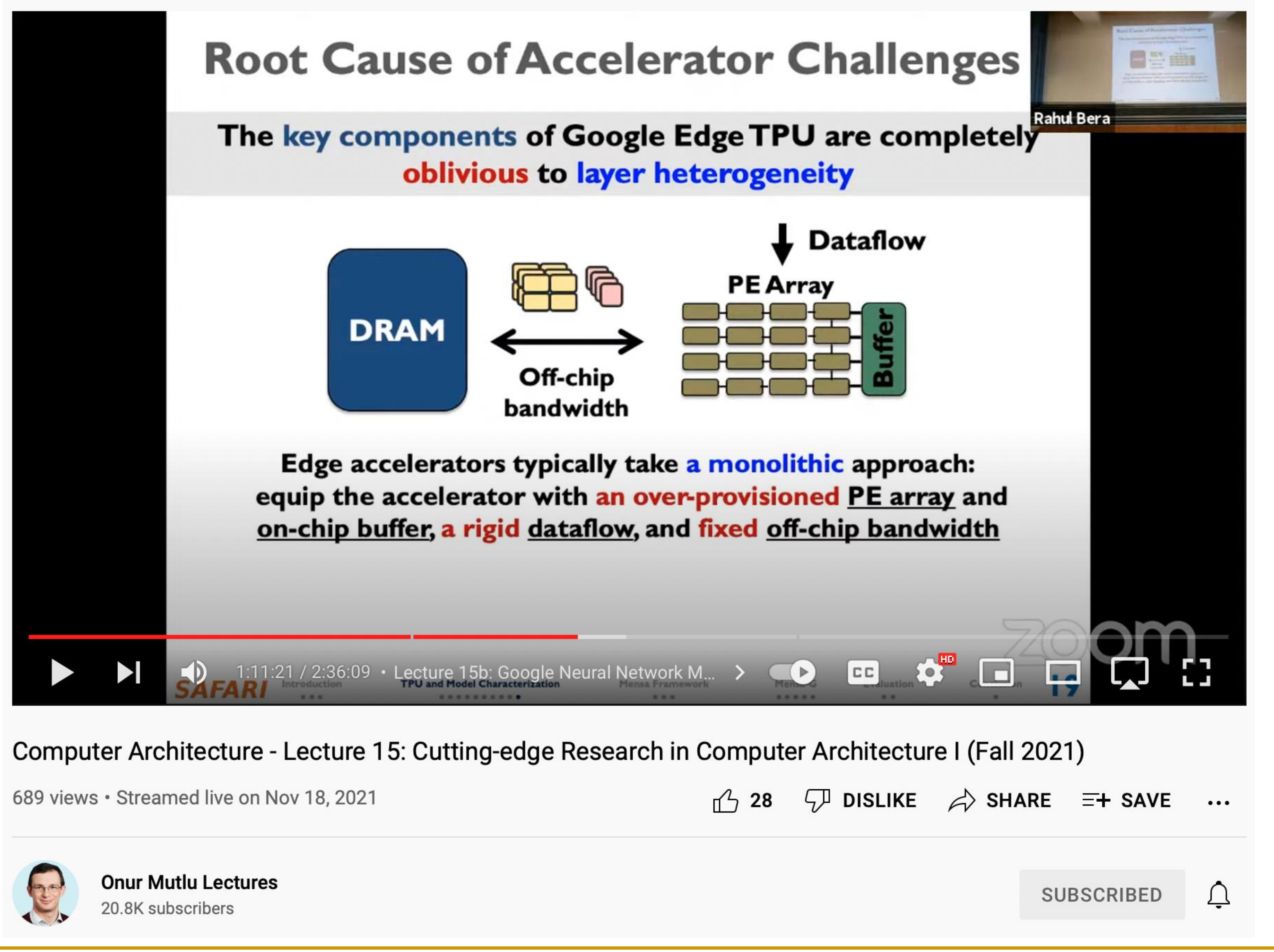
Unlike conventional SIMD, register contents are *not* private to each thread, but shared inside the warp

* M. A. Raihan, N. Goli and T. M. Aamodt, "Modeling Deep Learning Accelerator Enabled GPUs," ISPASS 2019.

Edge TPU: Baseline Accelerator

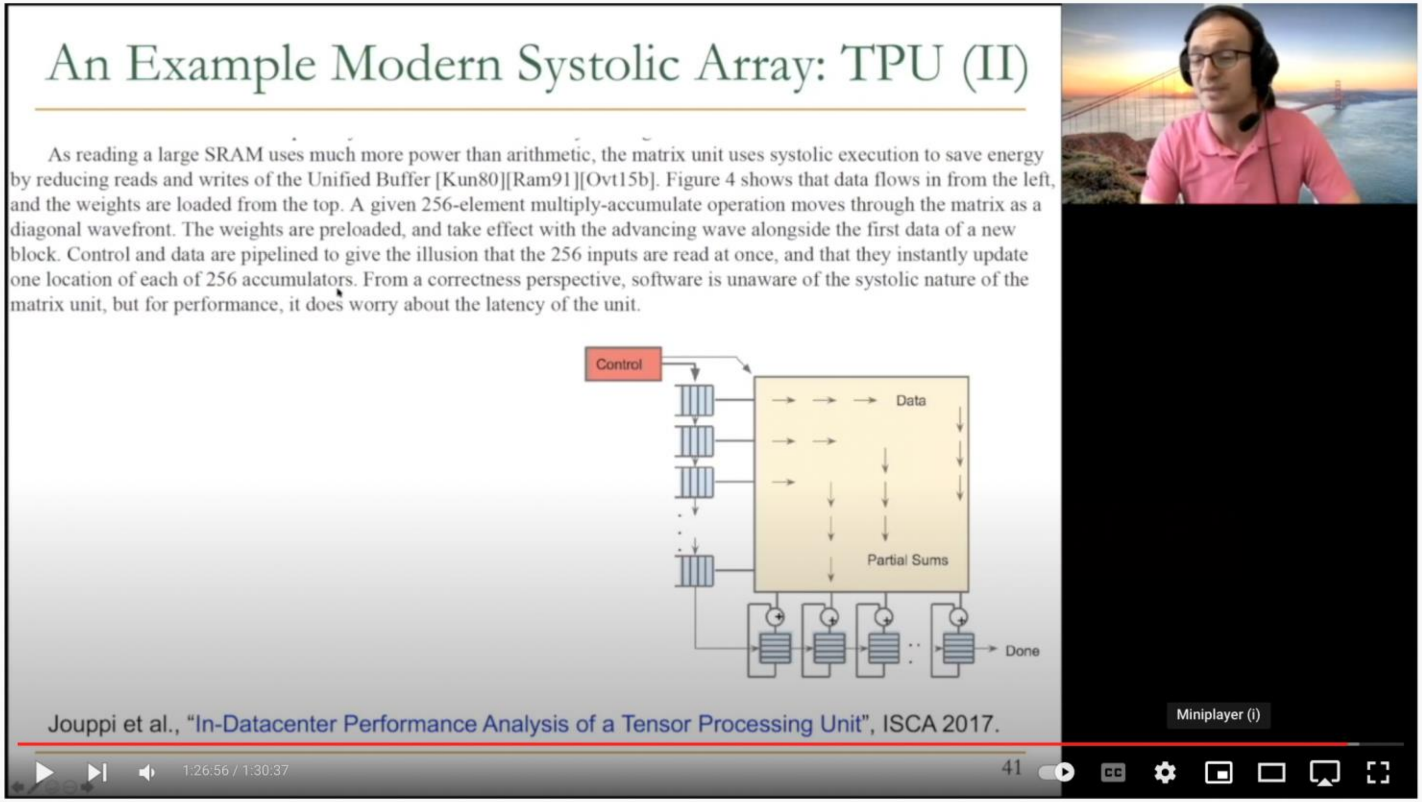


Research Lecture on Edge TPU



https://youtu.be/KPPfRRPENgQ?t=2999

Lecture 19b: Systolic Array Architectures



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

842 views • Premiered May 6, 2022

35 DISLIKE SHARE CLIP SAVE ...

Onur Mutlu Lectures
24.5K subscribers

SUBSCRIBED

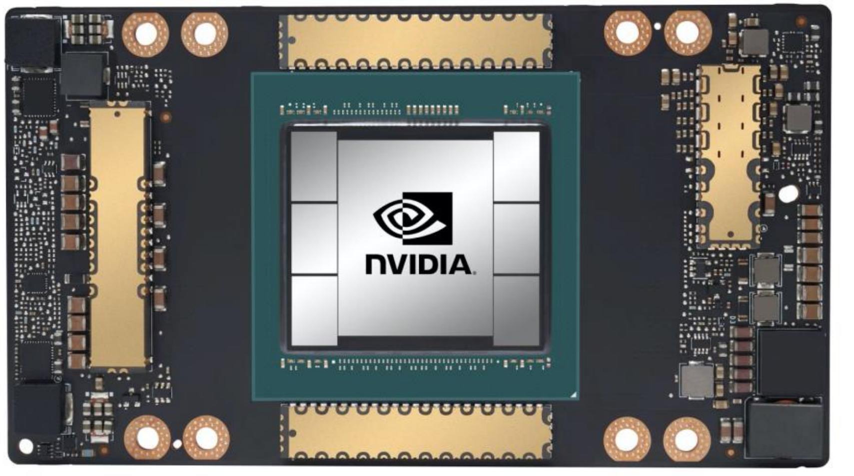
Digital Design and Computer Architecture, ETH Zürich, Spring 2022 (
https://safari.ethz.ch/digitaltechnik...)

Lecture 19a: VLIW Architectures
Lecture 19b: Systolic Array Architectures
Lecturer: Professor Onur Mutlu (https://people.inf.ethz.ch/omutlu/)
Date: May 6, 2022

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

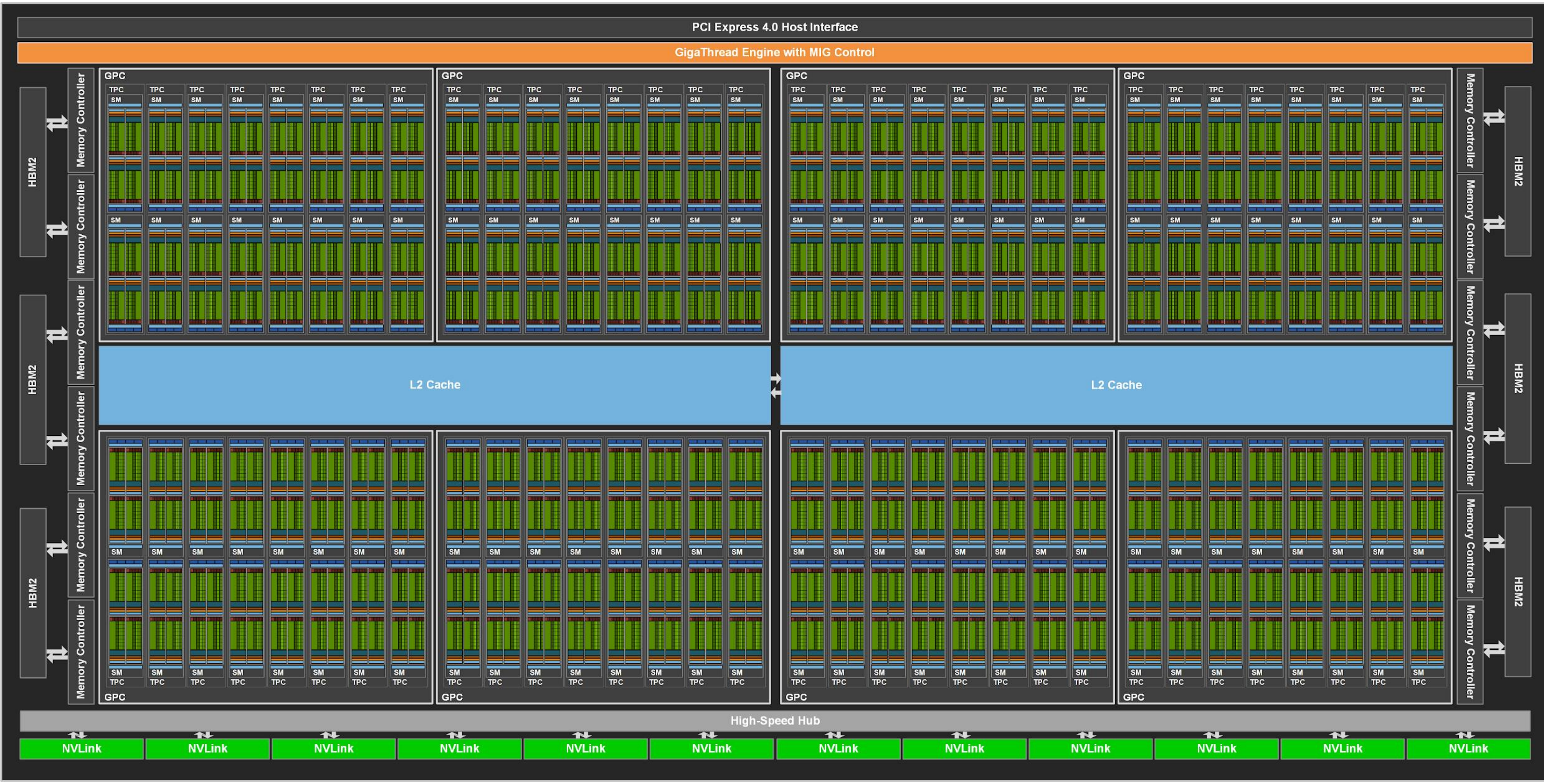
NVIDIA A100

- 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)

NVIDIA A100 Block Diagram



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

108 cores on the A100

(Up to 128 cores in the full-blown chip)

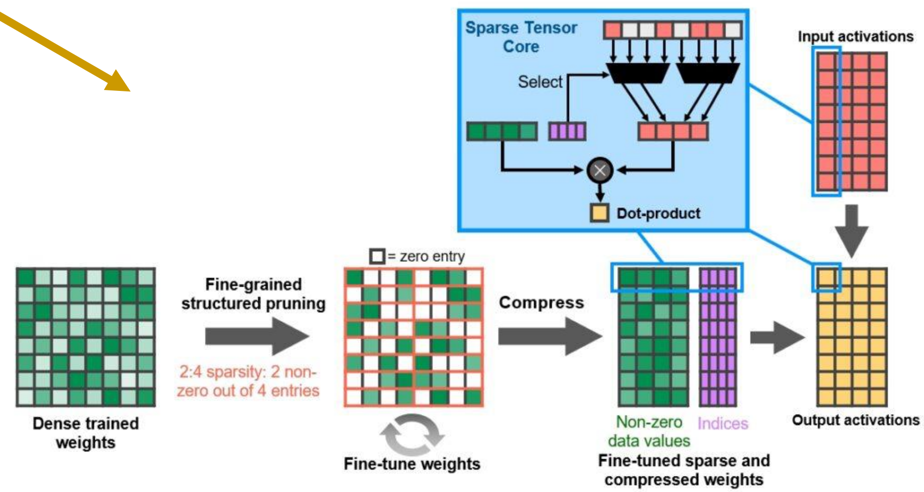
40MB L2 cache

NVIDIA A100 Core

19.5 TFLOPS Single Precision

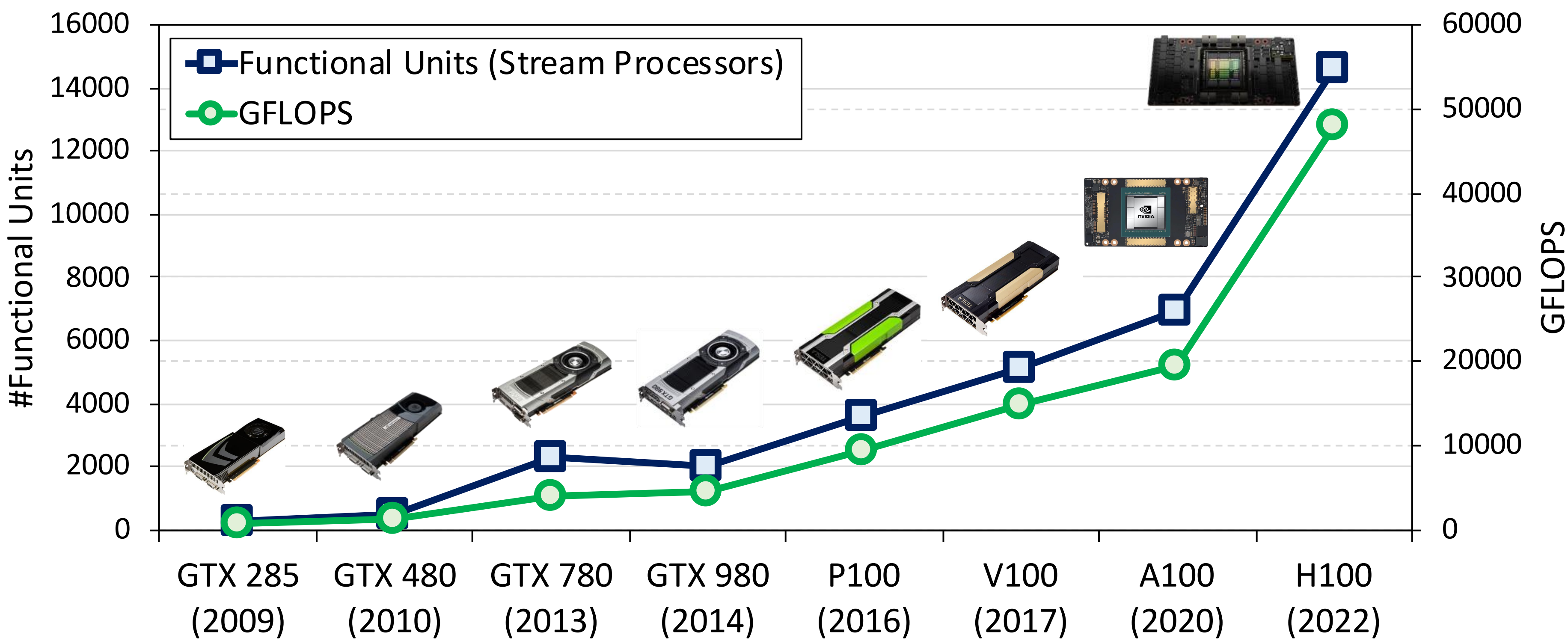
9.7 TFLOPS Double Precision

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

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

144 cores on the full GH100

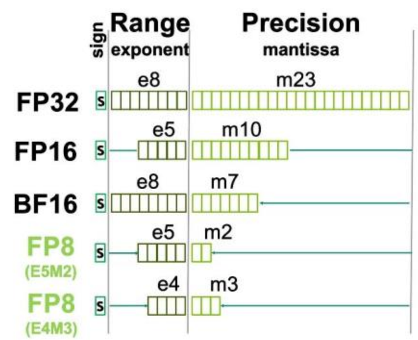
60MB L2 cache

NVIDIA H100 Core

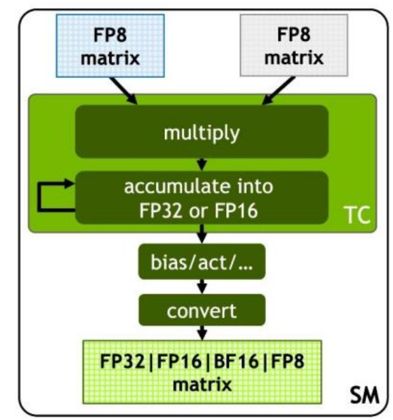
48 TFLOPS Single Precision*

24 TFLOPS Double Precision*

800 TFLOPS (FP16, Tensor Cores)*



Allocate 1 bit to either range or precision



Support for multiple accumulator and output types

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

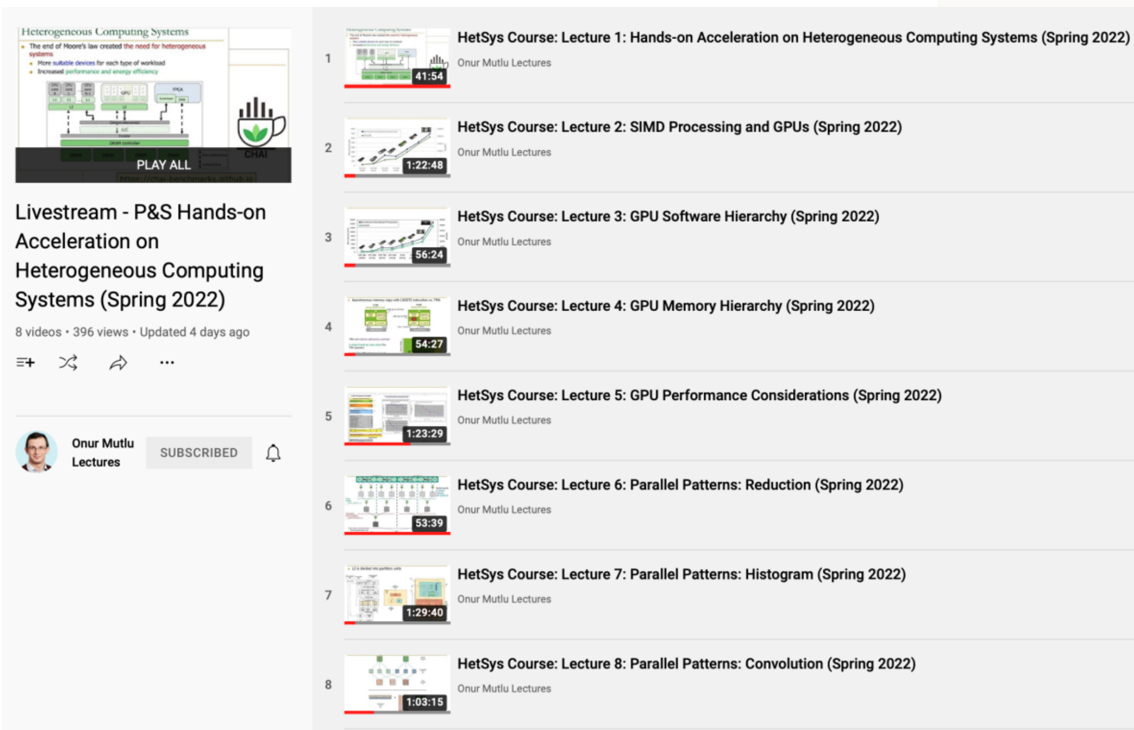
* Preliminary performance estimates

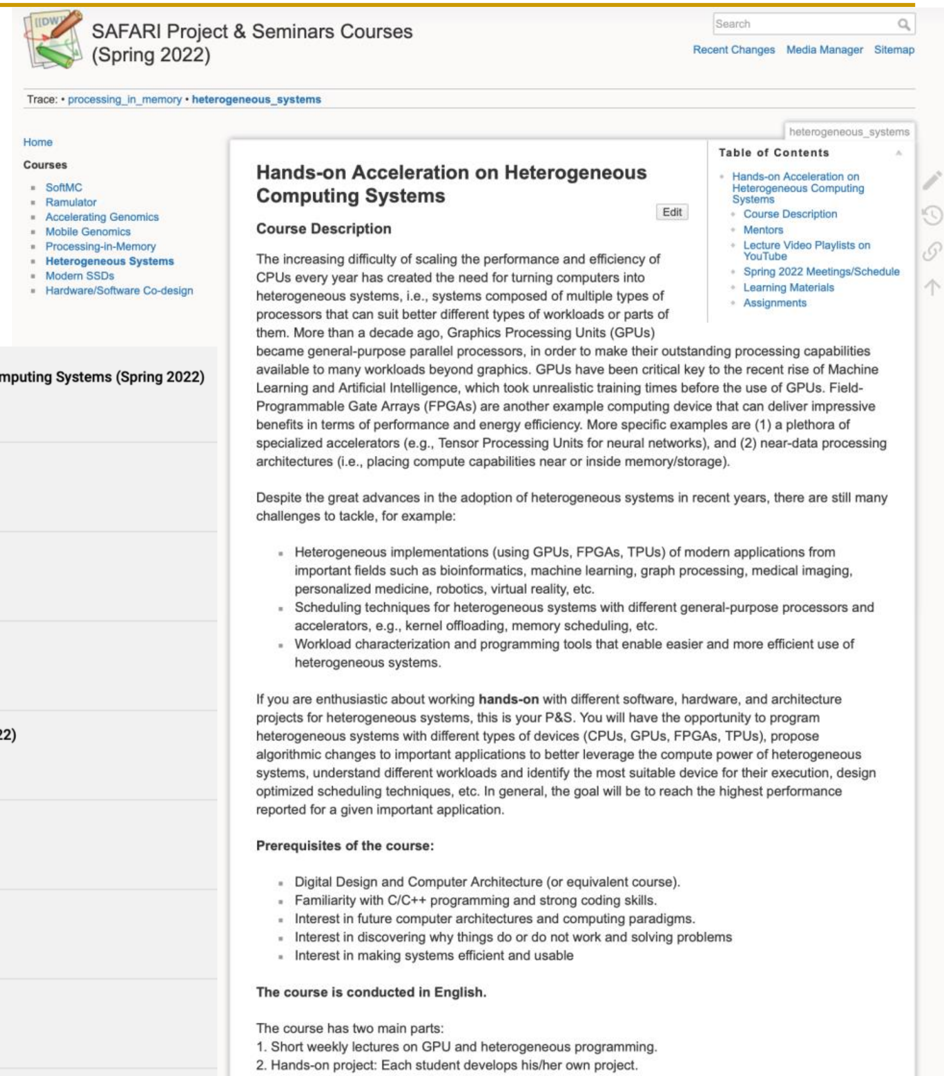
Food for Thought

- 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





SAFARI Project & Seminars Courses (Spring 2022)

Search | Recent Changes | Media Manager | Sitemap

Trace: • processing_in_memory • **heterogeneous_systems**

Home

Courses

- SoftMC
- Ramulator
- Accelerating Genomics
- Mobile Genomics
- Processing-in-Memory
- **Heterogeneous Systems**
- Modern SSDs
- Hardware/Software Co-design

Table of Contents

- Hands-on Acceleration on Heterogeneous Computing Systems
  - Course Description
  - Mentors
  - Lecture Video Playlists on YouTube
  - Spring 2022 Meetings/Schedule
  - Learning Materials
  - Assignments

Hands-on Acceleration on Heterogeneous Computing Systems

Course Description

The increasing difficulty of scaling the performance and efficiency of CPUs every year has created the need for turning computers into heterogeneous systems, i.e., systems composed of multiple types of processors that can suit better different types of workloads or parts of them. More than a decade ago, Graphics Processing Units (GPUs) became general-purpose parallel processors, in order to make their outstanding processing capabilities available to many workloads beyond graphics. GPUs have been critical key to the recent rise of Machine Learning and Artificial Intelligence, which took unrealistic training times before the use of GPUs. Field-Programmable Gate Arrays (FPGAs) are another example computing device that can deliver impressive benefits in terms of performance and energy efficiency. More specific examples are (1) a plethora of specialized accelerators (e.g., Tensor Processing Units for neural networks), and (2) near-data processing architectures (i.e., placing compute capabilities near or inside memory/storage).

Despite the great advances in the adoption of heterogeneous systems in recent years, there are still many challenges to tackle, for example:

- Heterogeneous implementations (using GPUs, FPGAs, TPUs) of modern applications from important fields such as bioinformatics, machine learning, graph processing, medical imaging, personalized medicine, robotics, virtual reality, etc.
- Scheduling techniques for heterogeneous systems with different general-purpose processors and accelerators, e.g., kernel offloading, memory scheduling, etc.
- Workload characterization and programming tools that enable easier and more efficient use of heterogeneous systems.

If you are enthusiastic about working **hands-on** with different software, hardware, and architecture projects for heterogeneous systems, this is your P&S. You will have the opportunity to program heterogeneous systems with different types of devices (CPUs, GPUs, FPGAs, TPUs), propose algorithmic changes to important applications to better leverage the compute power of heterogeneous systems, understand different workloads and identify the most suitable device for their execution, design optimized scheduling techniques, etc. In general, the goal will be to reach the highest performance reported for a given important application.

Prerequisites of the course:

- Digital Design and Computer Architecture (or equivalent course).
- Familiarity with C/C++ programming and strong coding skills.
- Interest in future computer architectures and computing paradigms.
- Interest in discovering why things do or do not work and solving problems
- Interest in making systems efficient and usable

The course is conducted in English.

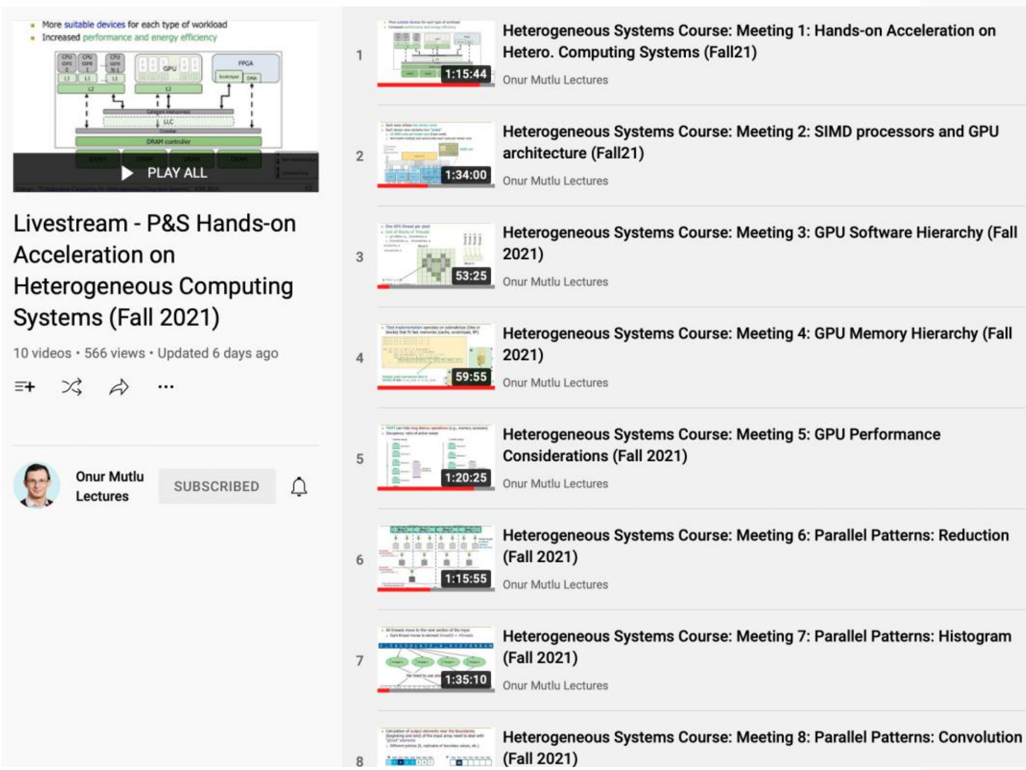
The course has two main parts:
1. Short weekly lectures on GPU and heterogeneous programming.
2. Hands-on project: Each student develops his/her own project.

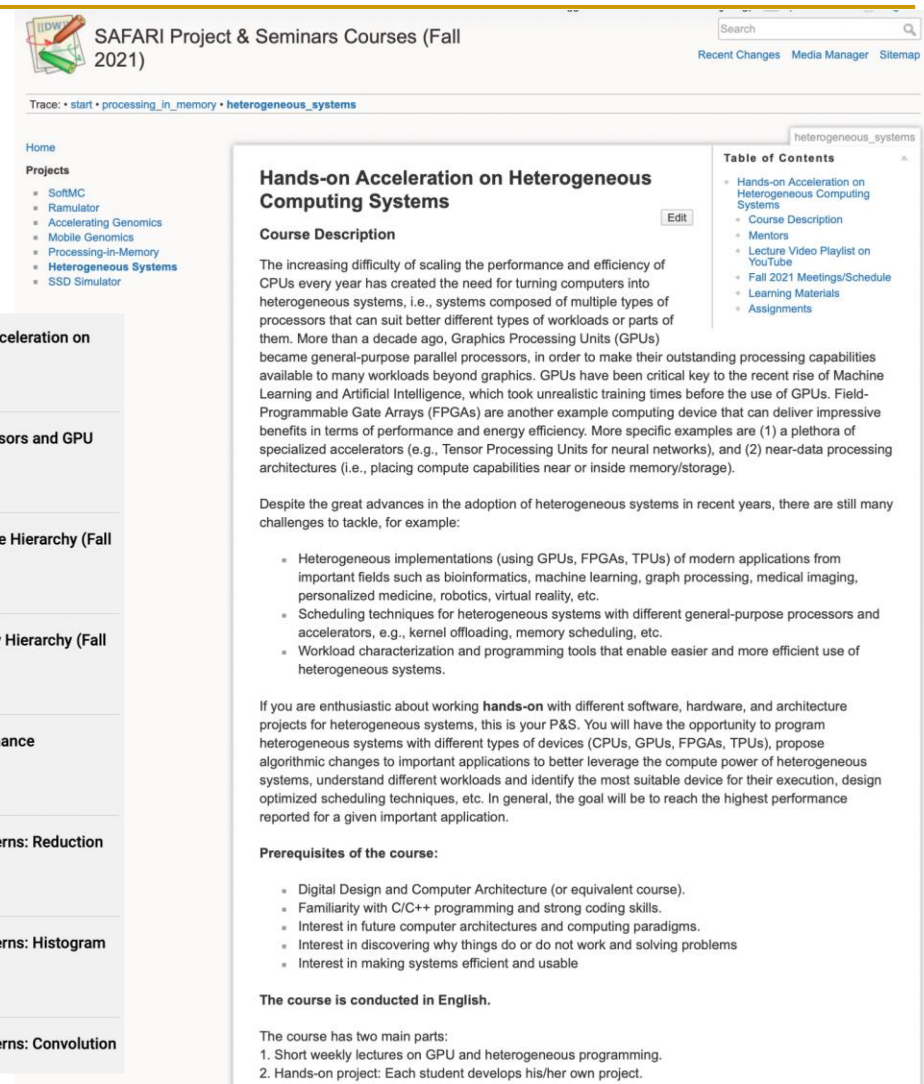
https://youtube.com/playlist?list=PL5Q2soXY2Zi9XrgXR38IM_FTjmY6h7Gzm

https://safari.ethz.ch/projects_and_seminars/spring2022/doku.php?id=heterogeneous_systems

Heterogeneous Systems Course (Fall 2021)

- Short weekly lectures
- Hands-on projects





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

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

Digital Design & Computer Arch.

Lecture 21: Graphics Processing Units

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich

Spring 2022

13 May 2022

Clarification of Some GPU Terms

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