Computer Architecture
Lecture 25: SIMD Processors and GPUs

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich
Fall 2022
05 January 2023

Agenda for This Lecture

- SIMD Processing
  - Vector and Array Processors

- Graphics Processing Units (GPUs)

Recommended Readings

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

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

Exploiting Data Parallelism: SIMD Processors and GPUs

SIMD Processing:
Exploiting Regular (Data) Parallelism

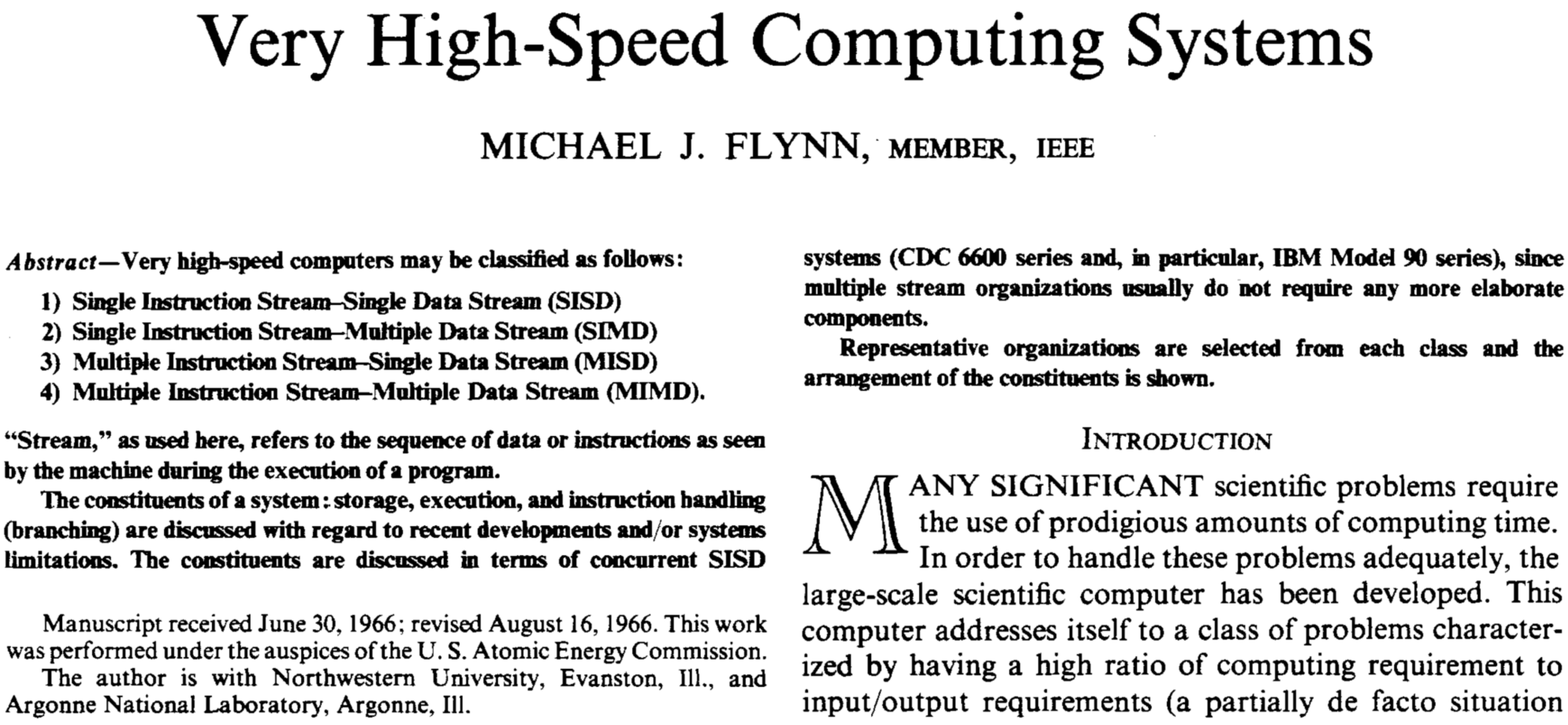
Flynn's Taxonomy of Computers

- 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

Flynn's Taxonomy of Computers

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



Very High-Speed Computing Systems

MICHAEL J. FLYNN, MEMBER, IEEE

Abstract—Very high-speed computers may be classified as follows:

1) Single Instruction Stream–Single Data Stream (SISD)
2) Single Instruction Stream–Multiple Data Stream (SIMD)
3) Multiple Instruction Stream–Single Data Stream (MISD)
4) Multiple Instruction Stream–Multiple Data Stream (MIMD).

"Stream," as used here, refers to the sequence of data or instructions as seen by the machine during the execution of a program.

The constituents of a system: storage, execution, and instruction handling (branching) are discussed with regard to recent developments and/or systems limitations. The constituents are discussed in terms of concurrent SISD systems (CDC 6600 series and, in particular, IBM Model 90 series), since multiple stream organizations usually do not require any more elaborate components.

Representative organizations are selected from each class and the arrangement of the constituents is shown.

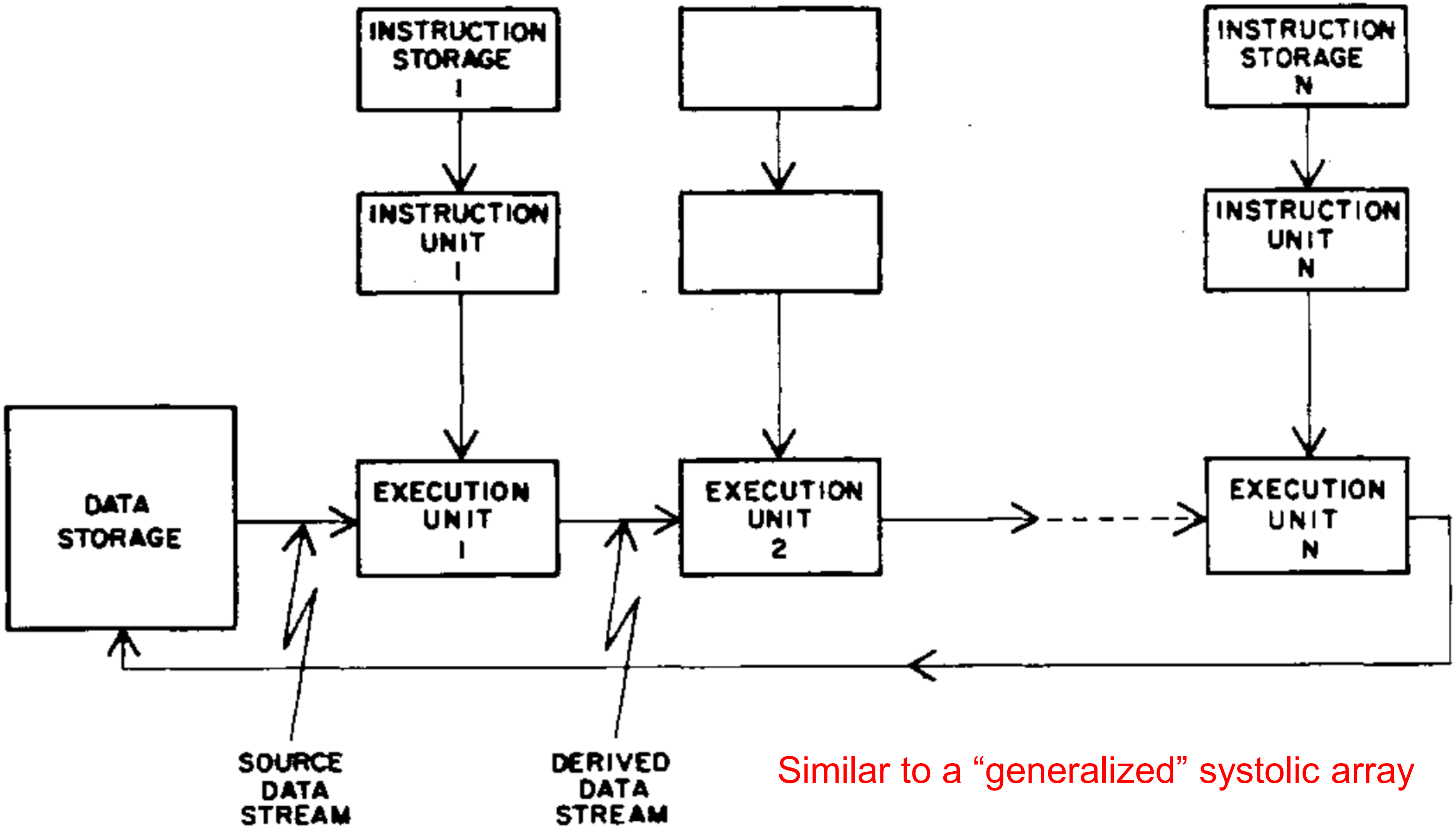
Manuscript received June 30, 1966; revised August 16, 1966. This work was performed under the auspices of the U. S. Atomic Energy Commission.

The author is with Northwestern University, Evanston, Ill., and Argonne National Laboratory, Argonne, Ill.

Introduction

MANY SIGNIFICANT scientific problems require the use of prodigious amounts of computing time. In order to handle these problems adequately, the large-scale scientific computer has been developed. This computer addresses itself to a class of problems characterized by having a high ratio of computing requirement to input/output requirements (a partially de facto situation

MISD Example from Flynn



Similar to a "generalized" systolic array

Lecture on Systolic Arrays

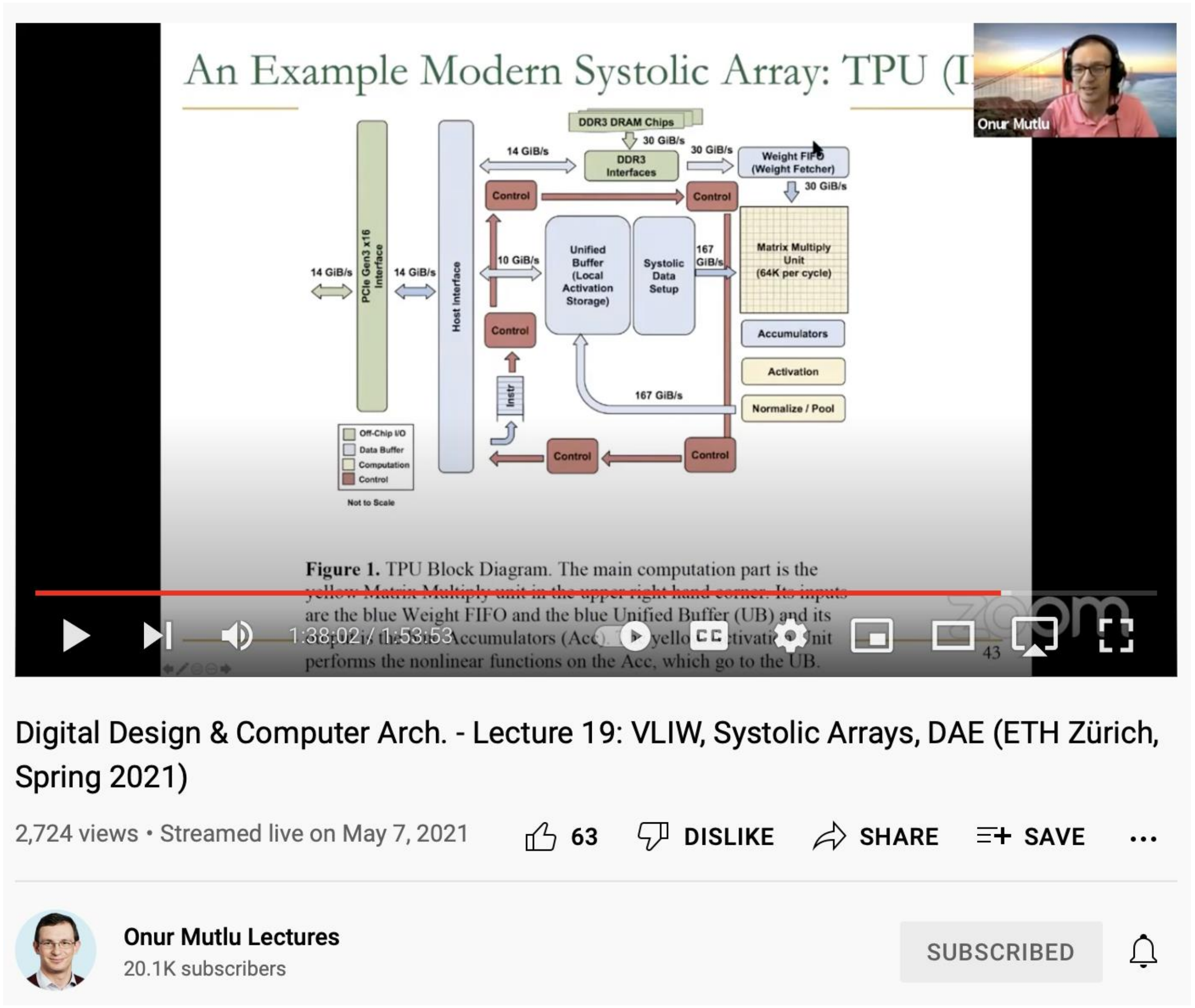


Figure 1. TPU Block Diagram. The main computation part is the yellow Matrix Multiply unit in the upper right hand corner. Its inputs are the blue Weight FIFO and the blue Unified Buffer (UB) and its output is the blue Accumulators (Acc). The yellow Activation Unit performs the nonlinear functions on the Acc, which go to the UB.

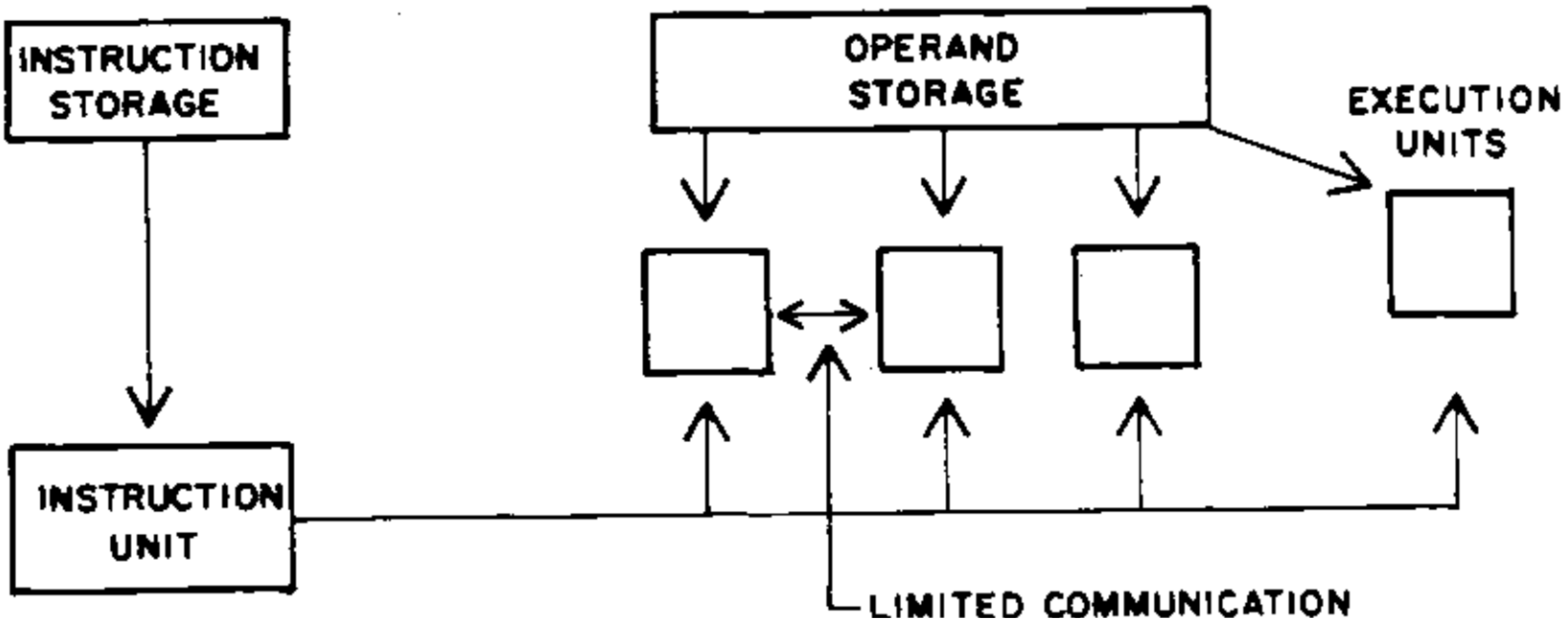
Digital Design & Computer Arch. - Lecture 19: VLIW, Systolic Arrays, DAE (ETH Zürich, Spring 2021)

2,724 views • Streamed live on May 7, 2021 63 DISLIKE SHARE SAVE

Onur Mutlu Lectures
20.1K subscribers
SUBSCRIBED

https://youtu.be/UtLy4Yagdys?t=2948

SIMD Example from Flynn



Similar to an "array processor"

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

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

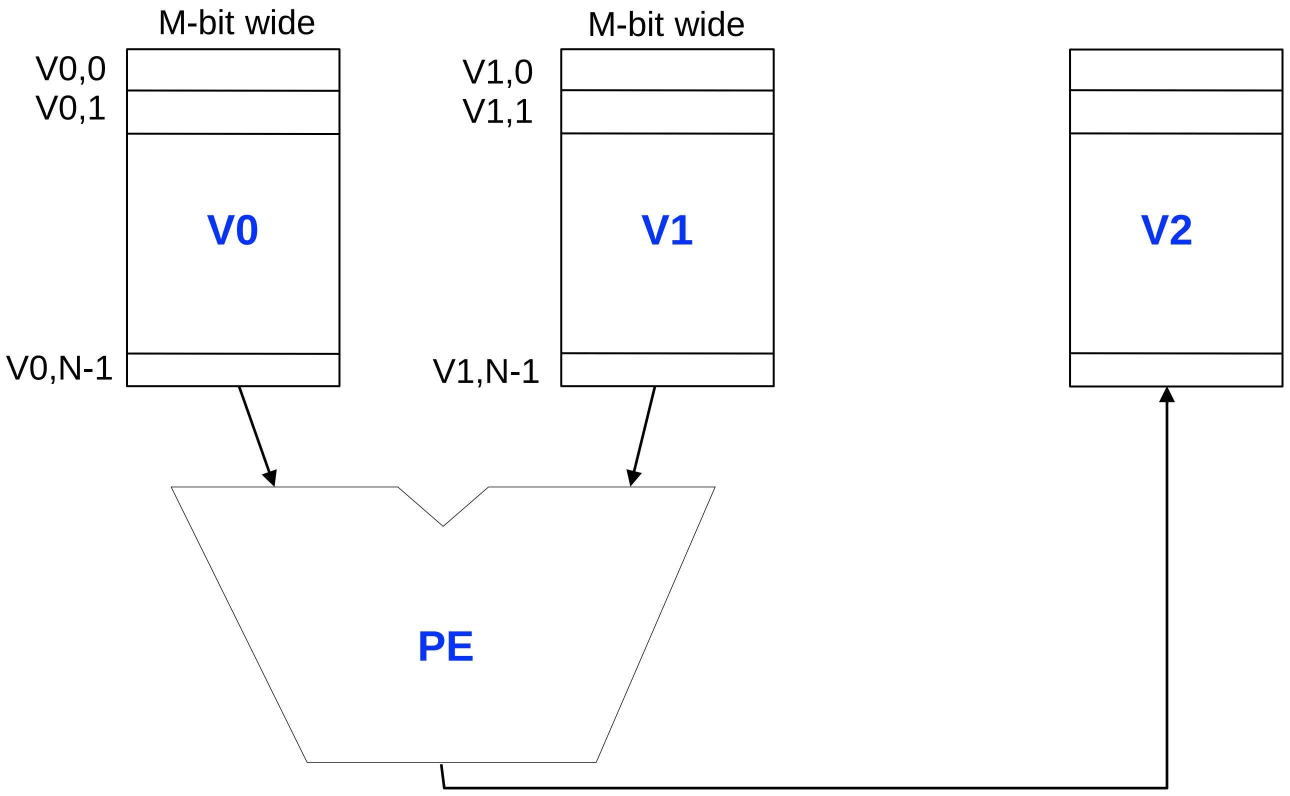
- Time-space duality

  - 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)

Storing Multiple Data Elements: Vector Registers

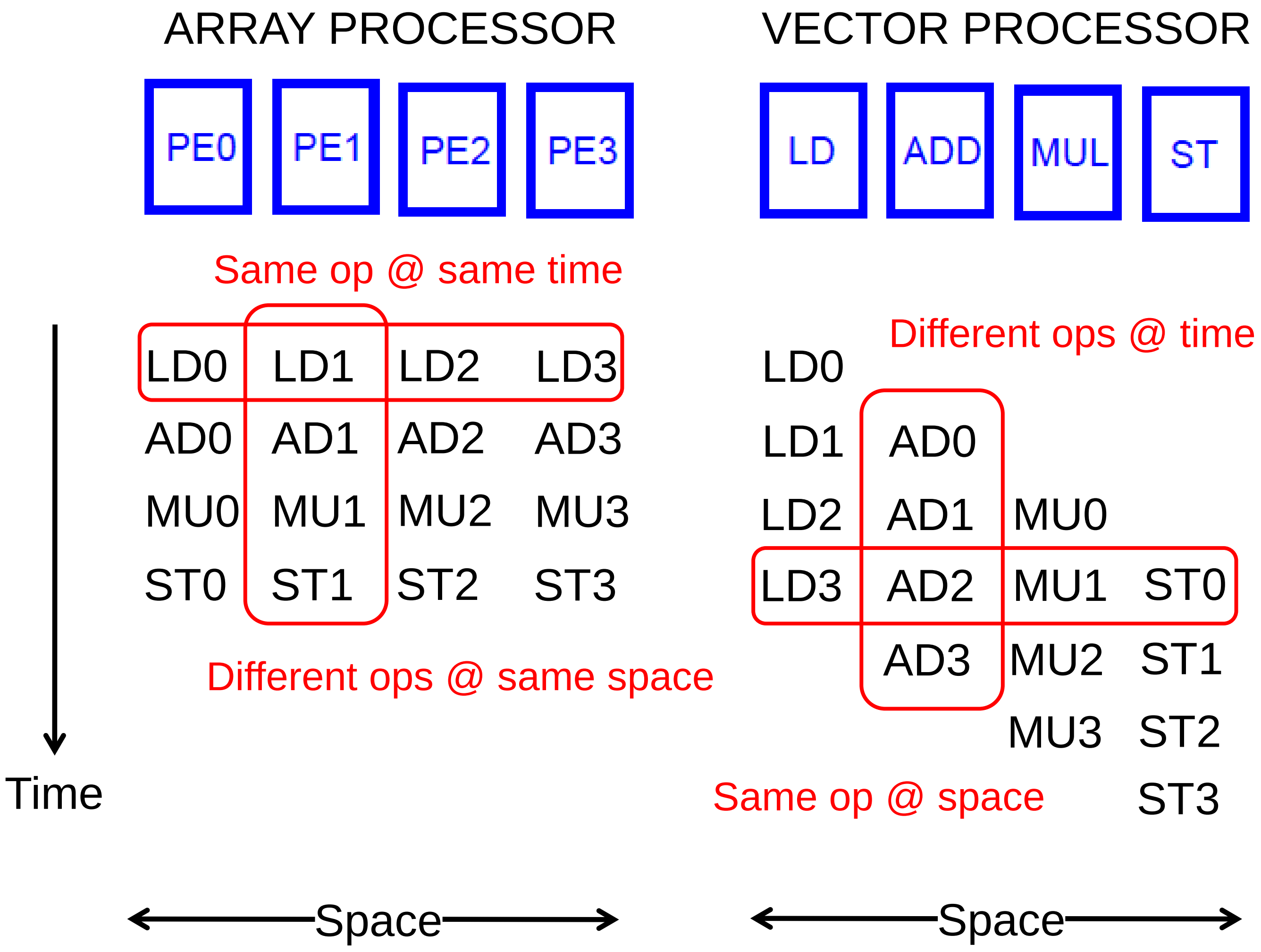
- Each vector data register holds N M-bit values
  - Each register stores a vector
  - Not a (single) scalar value as we saw before



Array vs. Vector Processors

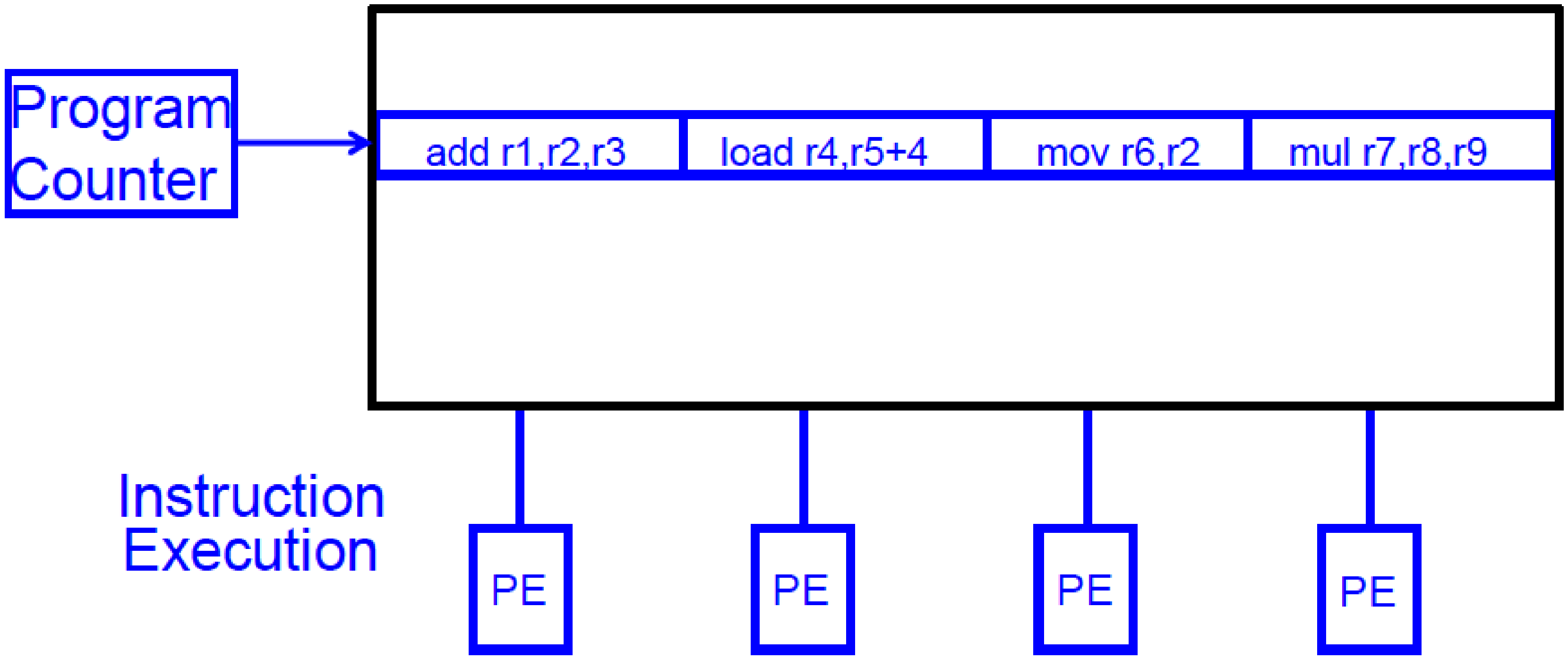
Instruction Stream

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



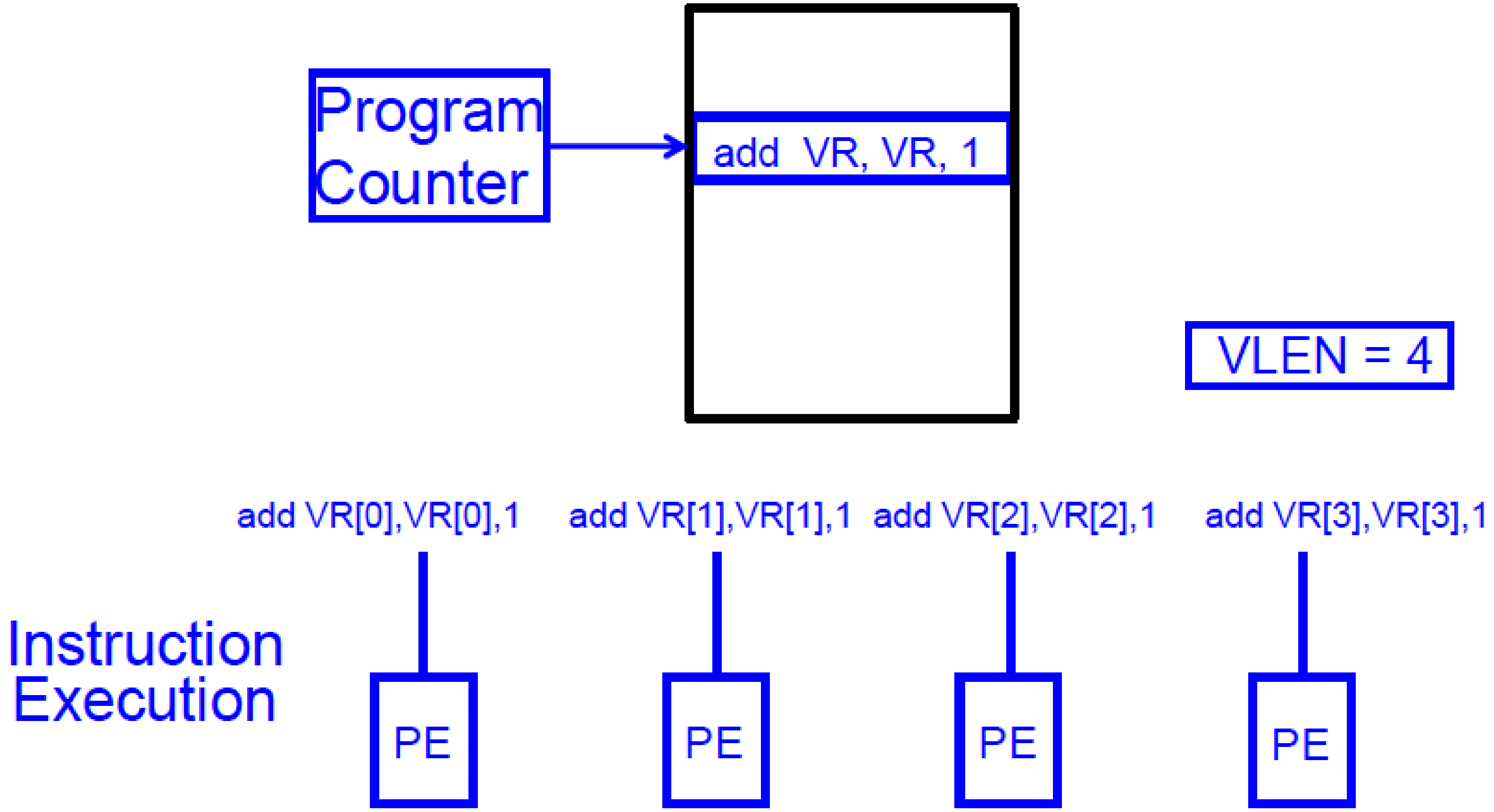
SIMD Array Processing vs. **VLIW**

- VLIW: **Multiple** independent **operations** packed together into a "long inst."

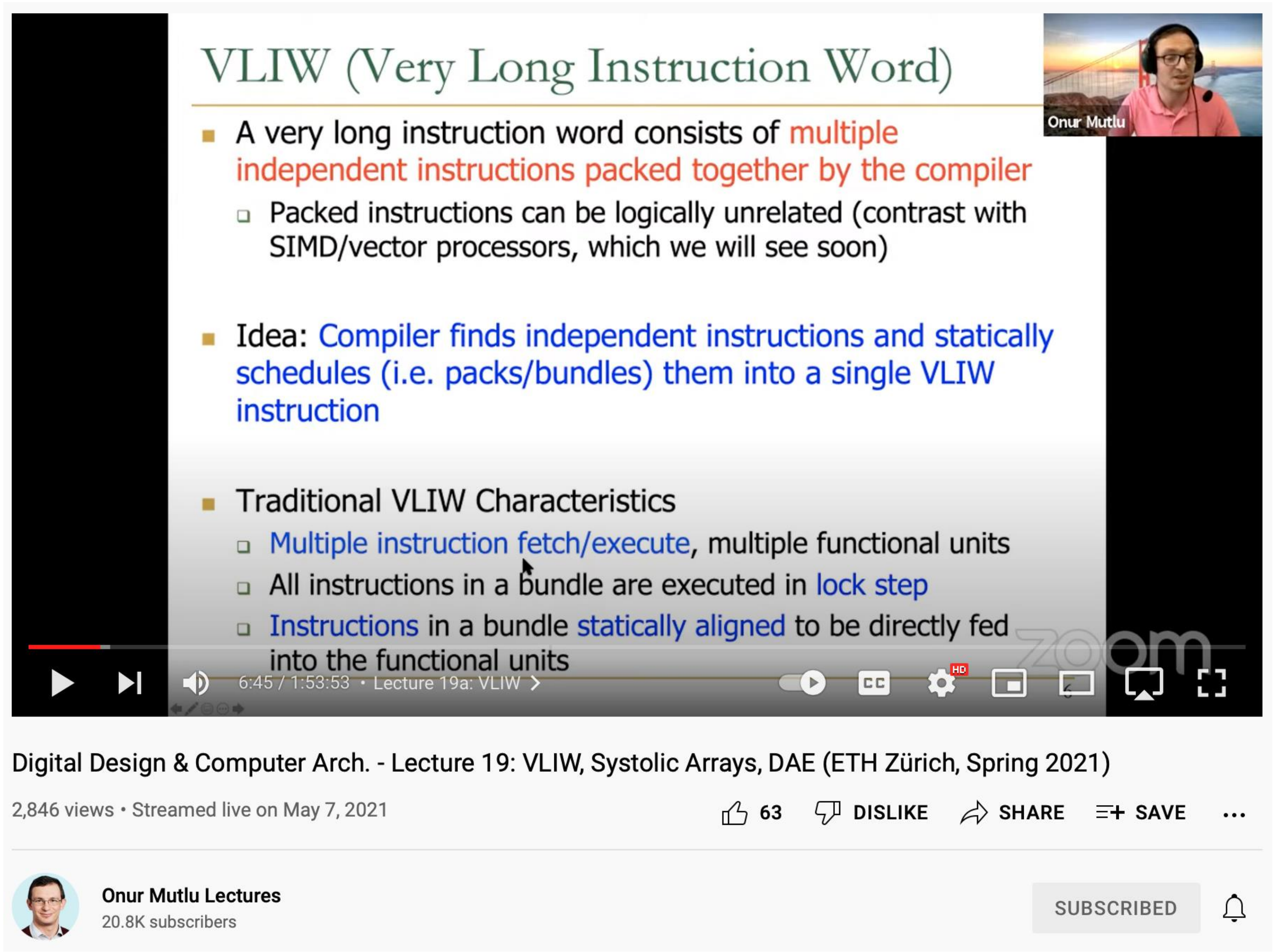


SIMD Array Processing vs. VLIW

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



Lecture on VLIW



VLIW (Very Long Instruction Word)

- A very long instruction word consists of multiple independent instructions packed together by the compiler
  - Packed instructions can be logically unrelated (contrast with SIMD/vector processors, which we will see soon)
- Idea: Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction
- Traditional VLIW Characteristics
  - Multiple instruction fetch/execute, multiple functional units
  - All instructions in a bundle are executed in lock step
  - Instructions in a bundle statically aligned to be directly fed into the functional units

Digital Design & Computer Arch. - Lecture 19: VLIW, Systolic Arrays, DAE (ETH Zürich, Spring 2021)

2,846 views • Streamed live on May 7, 2021

https://youtu.be/UtLy4Yagdys

Vector Processors (I)

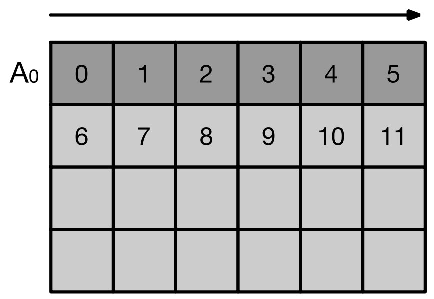
- A vector is a one-dimensional array of numbers
- Many scientific/commercial programs use vectors

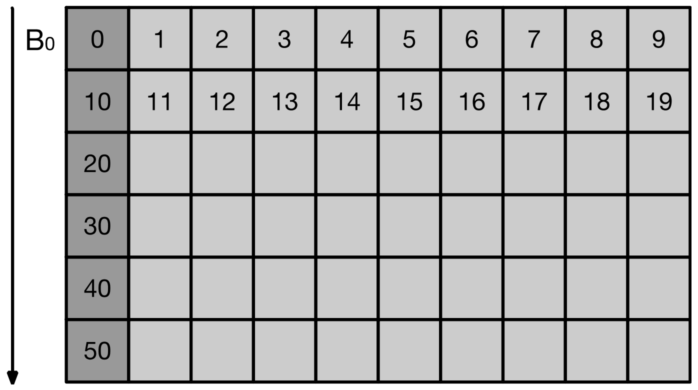
```
for (i = 0; i<=49; i++)
    C[i] = (A[i] + B[i]) / 2
```

- A vector processor is one whose instructions operate on vectors rather than scalar (single data) values
- Basic requirements
  - Need to load/store vectors → vector registers (contain vectors)
  - Need to operate on vectors of different lengths → vector length register (VLEN)
  - Elements of a vector might be stored apart from each other in memory → vector stride register (VSTR)
    - Stride: distance in memory between two elements of a vector

Vector Stride Example: Matrix Multiply

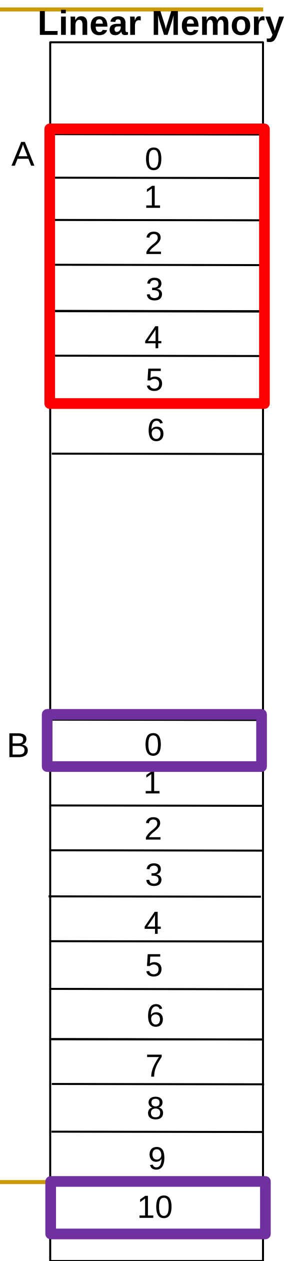
- A and B matrices, both stored in memory in row-major order





$A_{4x6} \, B_{6x10} \rightarrow C_{4x10}$

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

Vector Processor Advantages

+ No dependencies within a vector
  - Pipelining & parallelization work really well
  - Can have very deep pipelines (without the penalty of deep pipelines)

+ Each instruction generates a lot of work (i.e., operations)
  - Reduces instruction fetch bandwidth requirements
  - Amortizes instruction fetch and control overhead over many data

    --> Leads to high energy efficiency per operation

+ No need to explicitly code loops
  - Fewer branches in the instruction sequence

+ Highly regular memory access pattern

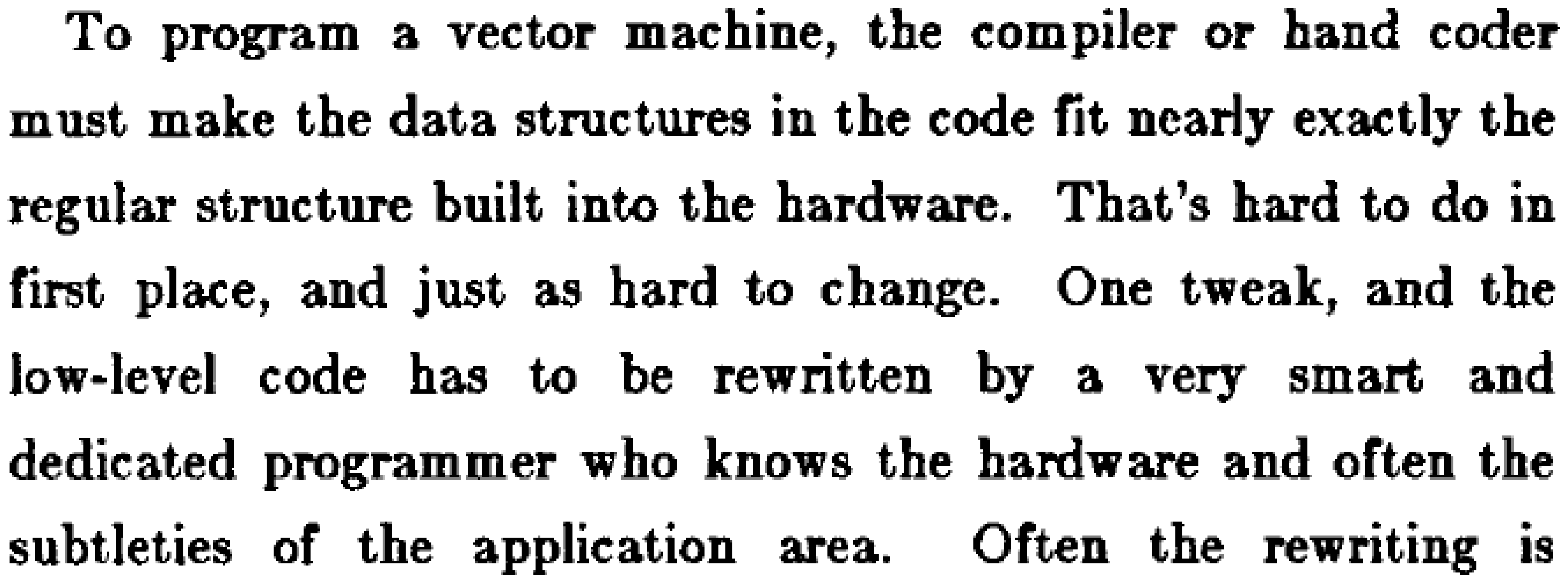
Vector Processor Disadvantages

-- Works (only) if parallelism is regular (data/SIMD parallelism)

  ++ Vector operations

  -- Very inefficient if parallelism is irregular

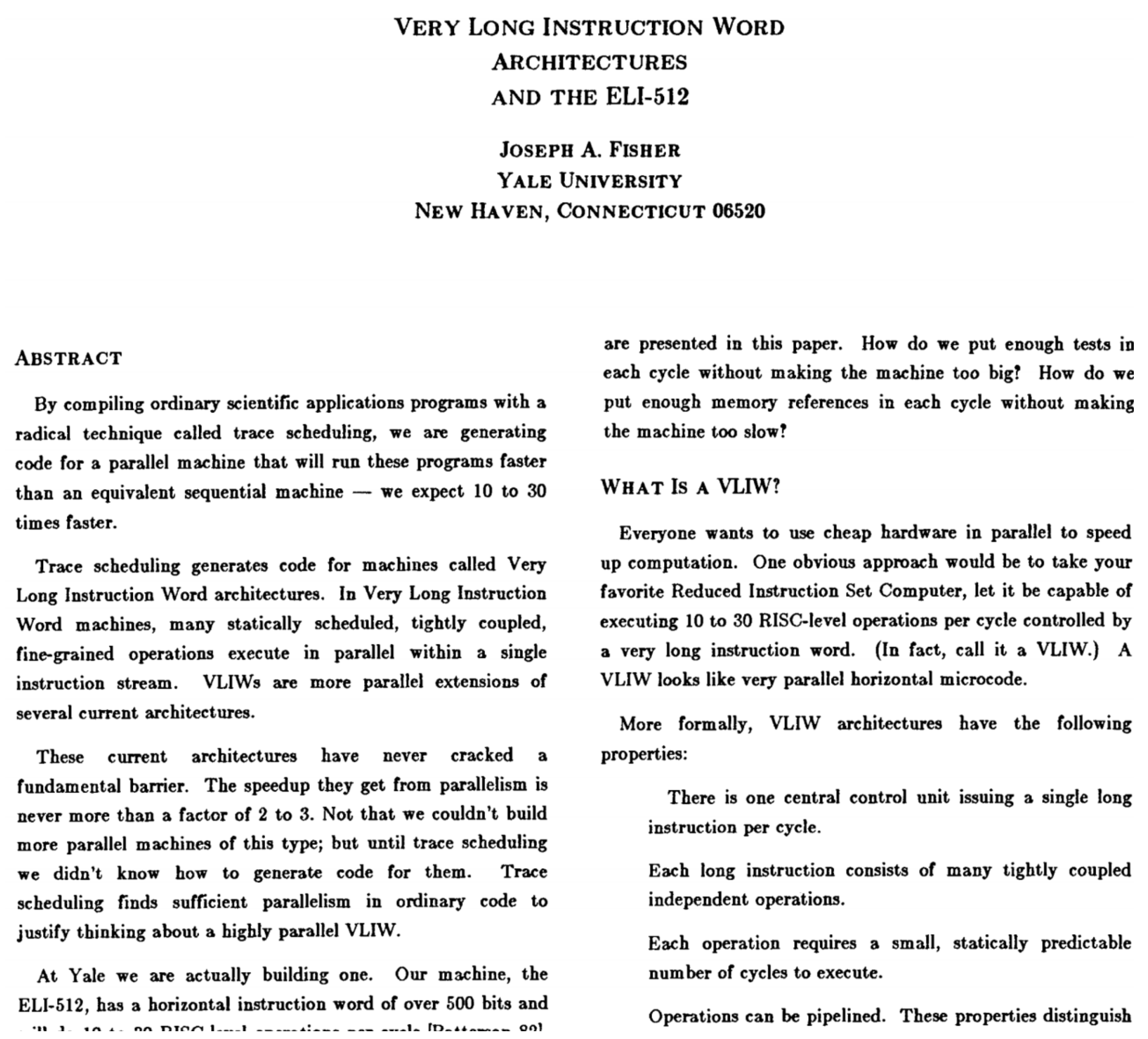
    -- How about searching for a key in a linked list?



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.

Recommended Paper



VERY LONG INSTRUCTION WORD ARCHITECTURES AND THE ELI-512

JOSEPH A. FISHER
YALE UNIVERSITY
NEW HAVEN, CONNECTICUT 06520

ABSTRACT

By compiling ordinary scientific applications programs with a radical technique called trace scheduling, we are generating code for a parallel machine that will run these programs faster than an equivalent sequential machine — we expect 10 to 30 times faster.

Trace scheduling generates code for machines called Very Long Instruction Word architectures. In Very Long Instruction Word machines, many statically scheduled, tightly coupled, fine-grained operations execute in parallel within a single instruction stream. VLIWs are more parallel extensions of several current architectures.

These current architectures have never cracked a fundamental barrier. The speedup they get from parallelism is never more than a factor of 2 to 3. Not that we couldn't build more parallel machines of this type; but until trace scheduling we didn't know how to generate code for them. Trace scheduling finds sufficient parallelism in ordinary code to justify thinking about a highly parallel VLIW.

At Yale we are actually building one. Our machine, the ELI-512, has a horizontal instruction word of over 500 bits and

are presented in this paper. How do we put enough tests in each cycle without making the machine too big? How do we put enough memory references in each cycle without making the machine too slow?

WHAT IS A VLIW?

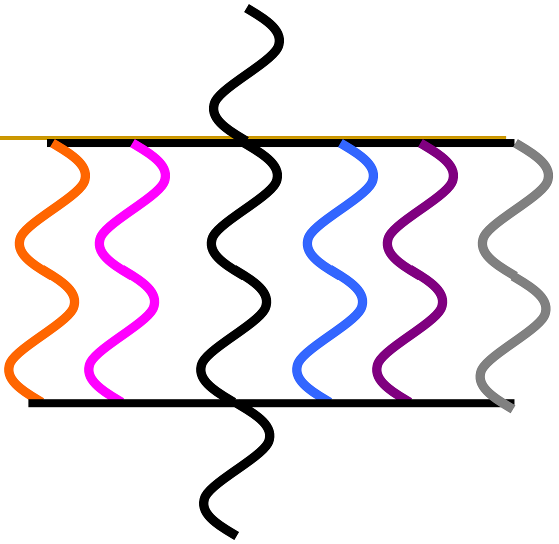
Everyone wants to use cheap hardware in parallel to speed up computation. One obvious approach would be to take your favorite Reduced Instruction Set Computer, let it be capable of executing 10 to 30 RISC-level operations per cycle controlled by a very long instruction word. (In fact, call it a VLIW.) A VLIW looks like very parallel horizontal microcode.

More formally, VLIW architectures have the following properties:

> There is one central control unit issuing a single long instruction per cycle.
>
> Each long instruction consists of many tightly coupled independent operations.
>
> Each operation requires a small, statically predictable number of cycles to execute.
>
> Operations can be pipelined. These properties distinguish

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

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

Vector Processor Limitations

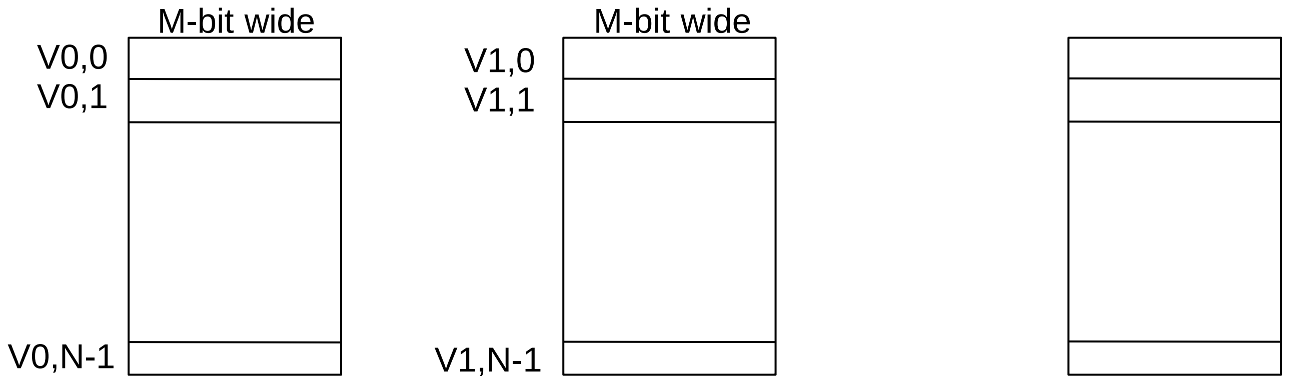
-- Memory (bandwidth) can easily become a bottleneck, especially if

1. compute/memory operation balance is not maintained
2. data is not mapped appropriately to memory banks

Vector Processing in More Depth

Vector Registers

- Each vector data register holds N M-bit values
- Vector control registers: VLEN, VSTR, VMASK
- Maximum VLEN can be N
  - Maximum number of elements stored in a vector register
- Vector Mask Register (VMASK)
  - Indicates which elements of vector to operate on
  - Set by vector test instructions
    - e.g., VMASK[i] = (V_k[i] == 0)

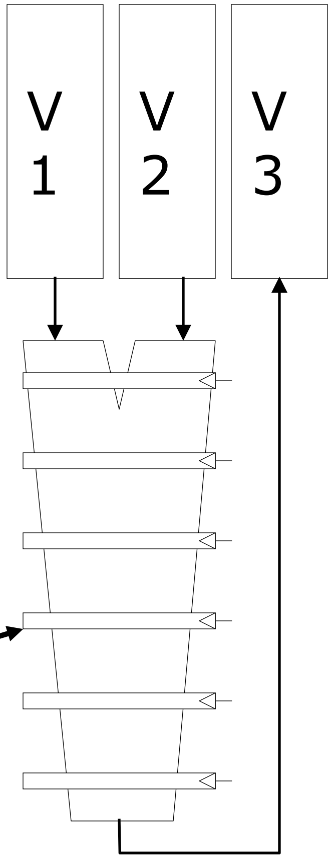


Vector Functional Units

- Use a deep pipeline to execute element operations

  → fast clock cycle

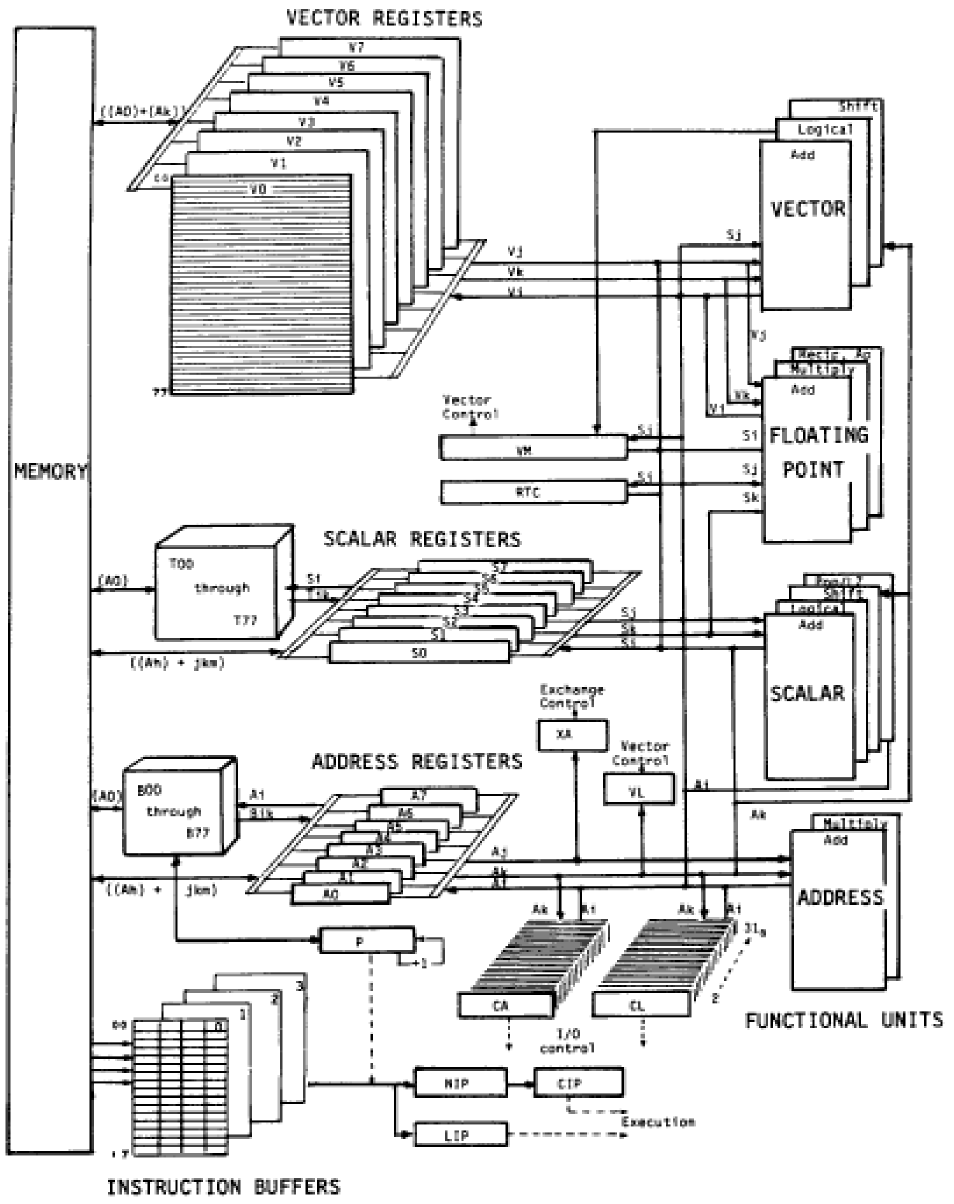
- Control of deep pipeline is simple because elements in vector are independent



Six stage multiply pipeline

V1 * V2 → V3

Vector Machine Organization (CRAY-1)

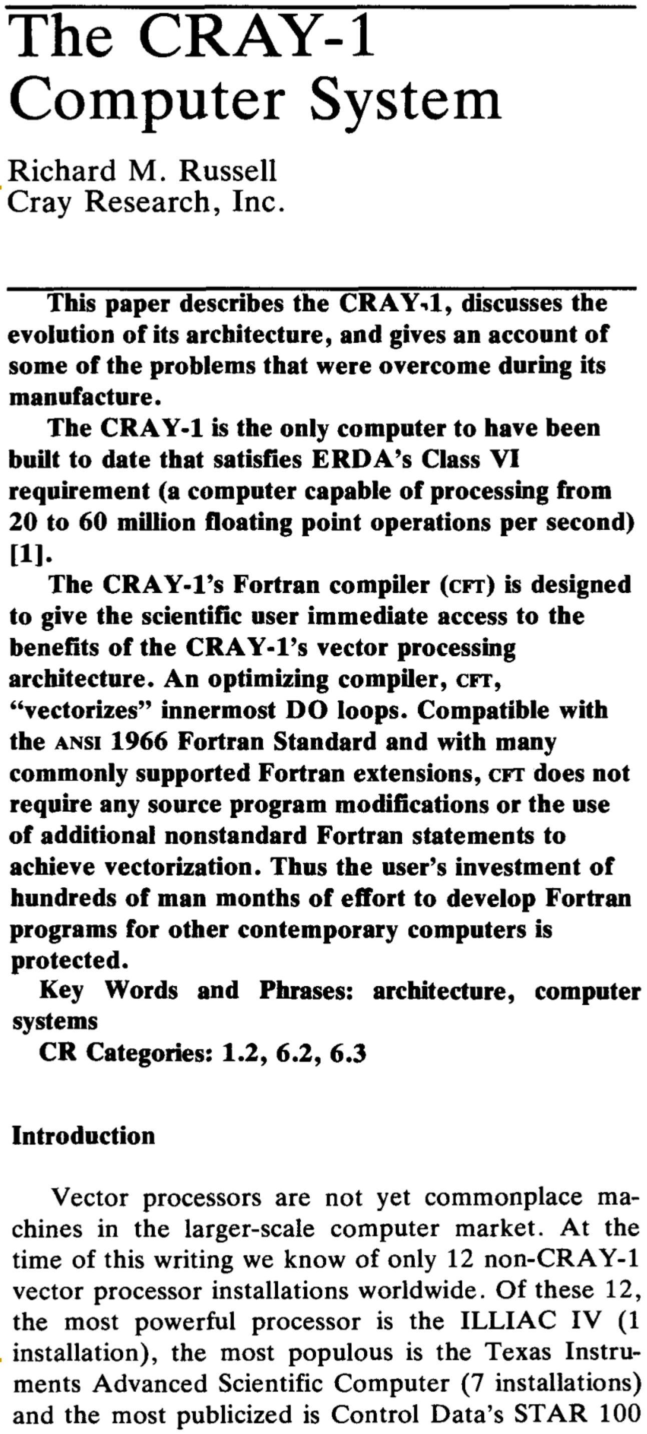


- CRAY-1
- Russell, "The CRAY-1 computer system," CACM 1978.

- Scalar and vector modes
- 8 64-element vector registers
- 64 bits per element
- 16 memory banks
- 8 64-bit scalar registers
- 8 24-bit address registers

Recommended Paper

Russell,
"The CRAY-1 computer system,"
CACM 1978.



The CRAY-1 Computer System

Richard M. Russell
Cray Research, Inc.

This paper describes the CRAY-1, discusses the evolution of its architecture, and gives an account of some of the problems that were overcome during its manufacture.

The CRAY-1 is the only computer to have been built to date that satisfies ERDA's Class VI requirement (a computer capable of processing from 20 to 60 million floating point operations per second) [1].

The CRAY-1's Fortran compiler (CFT) is designed to give the scientific user immediate access to the benefits of the CRAY-1's vector processing architecture. An optimizing compiler, CFT, "vectorizes" innermost DO loops. Compatible with the ANSI 1966 Fortran Standard and with many commonly supported Fortran extensions, CFT does not require any source program modifications or the use of additional nonstandard Fortran statements to achieve vectorization. Thus the user's investment of hundreds of man months of effort to develop Fortran programs for other contemporary computers is protected.

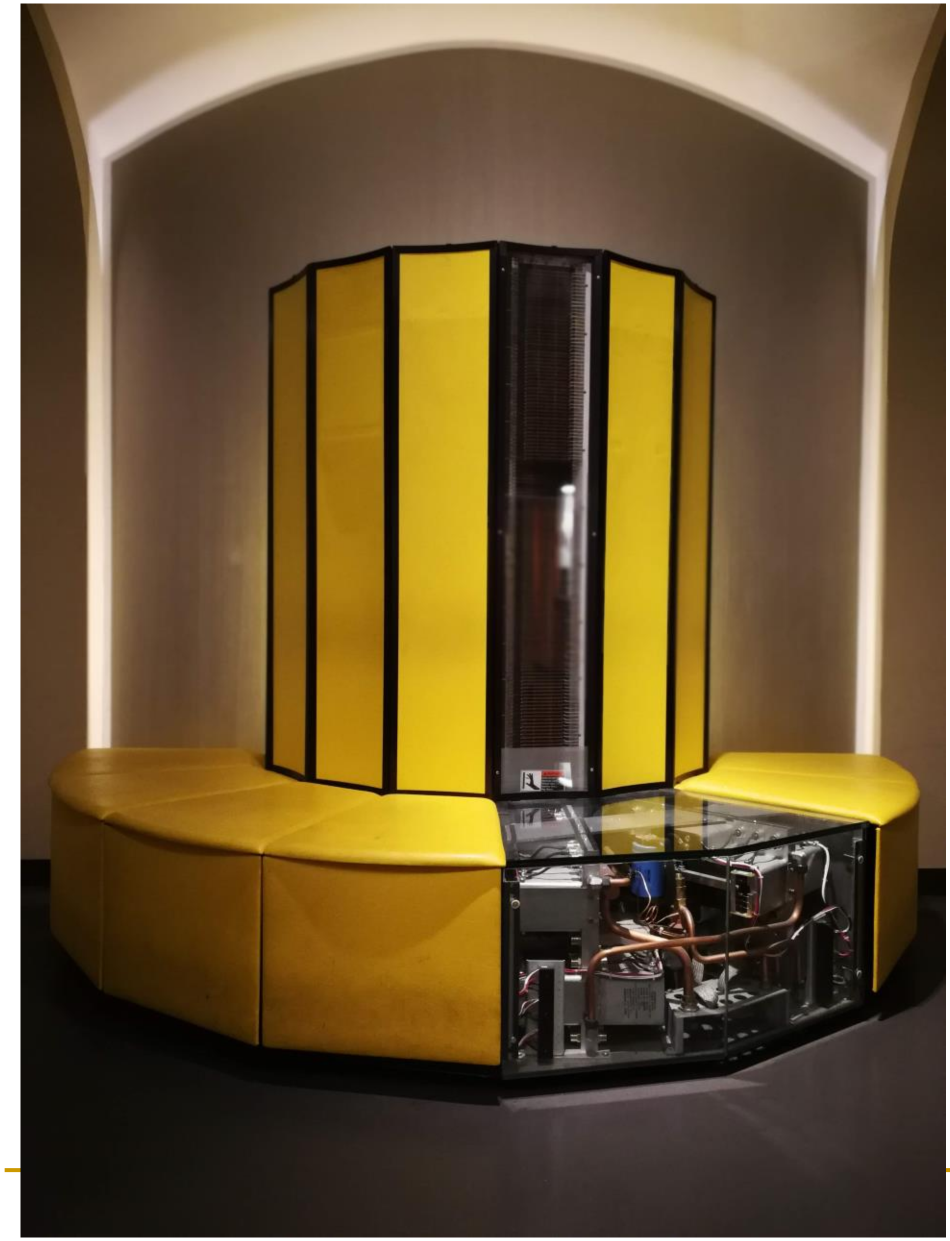
Key Words and Phrases: architecture, computer systems

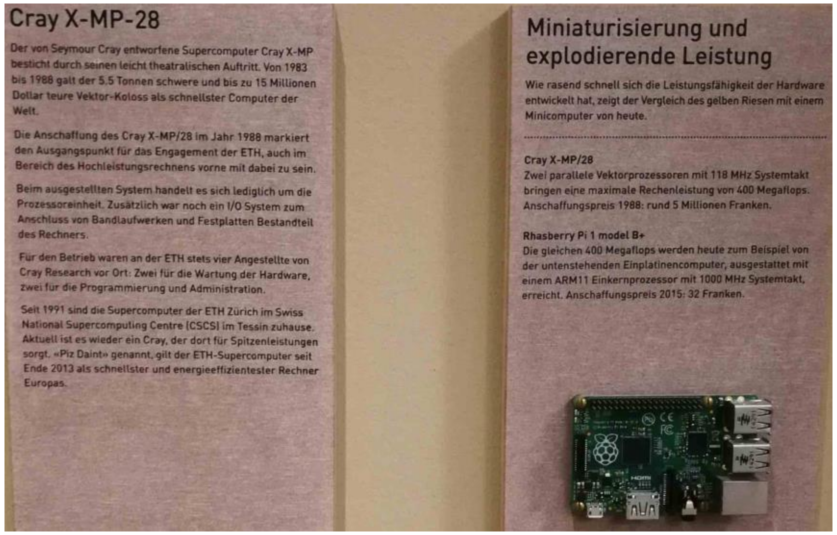
CR Categories: 1.2, 6.2, 6.3

Introduction

Vector processors are not yet commonplace machines in the larger-scale computer market. At the time of this writing we know of only 12 non-CRAY-1 vector processor installations worldwide. Of these 12, the most powerful processor is the ILLIAC IV (1 installation), the most populous is the Texas Instruments Advanced Scientific Computer (7 installations) and the most publicized is Control Data's STAR 100

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





Cray X-MP-28

Der von Seymour Cray entworfene Supercomputer Cray X-MP besticht durch seinen leicht theatralischen Auftritt. Von 1983 bis 1988 galt der 5.5 Tonnen schwere und bis zu 15 Millionen Dollar teure Vektor-Koloss als schnellster Computer der Welt.

Die Anschaffung des Cray X-MP/28 im Jahr 1988 markiert den Ausgangspunkt für das Engagement der ETH, auch im Bereich des Hochleistungsrechnens vorne mit dabei zu sein.

Beim ausgestellten System handelt es sich lediglich um die Prozessoreinheit. Zusätzlich war noch ein I/O System zum Anschluss von Bandlaufwerken und Festplatten Bestandteil des Rechners.

Für den Betrieb waren an der ETH stets vier Angestellte von Cray Research vor Ort: Zwei für die Wartung der Hardware, zwei für die Programmierung und Administration.

Seit 1991 sind die Supercomputer der ETH Zürich im Swiss National Supercomputing Centre (CSCS) im Tessin zuhause. Aktuell ist es wieder ein Cray, der dort für Spitzenleistungen sorgt. «Piz Daint» genannt, gilt der ETH-Supercomputer seit Ende 2013 als schnellster und energieeffizientester Rechner Europas.

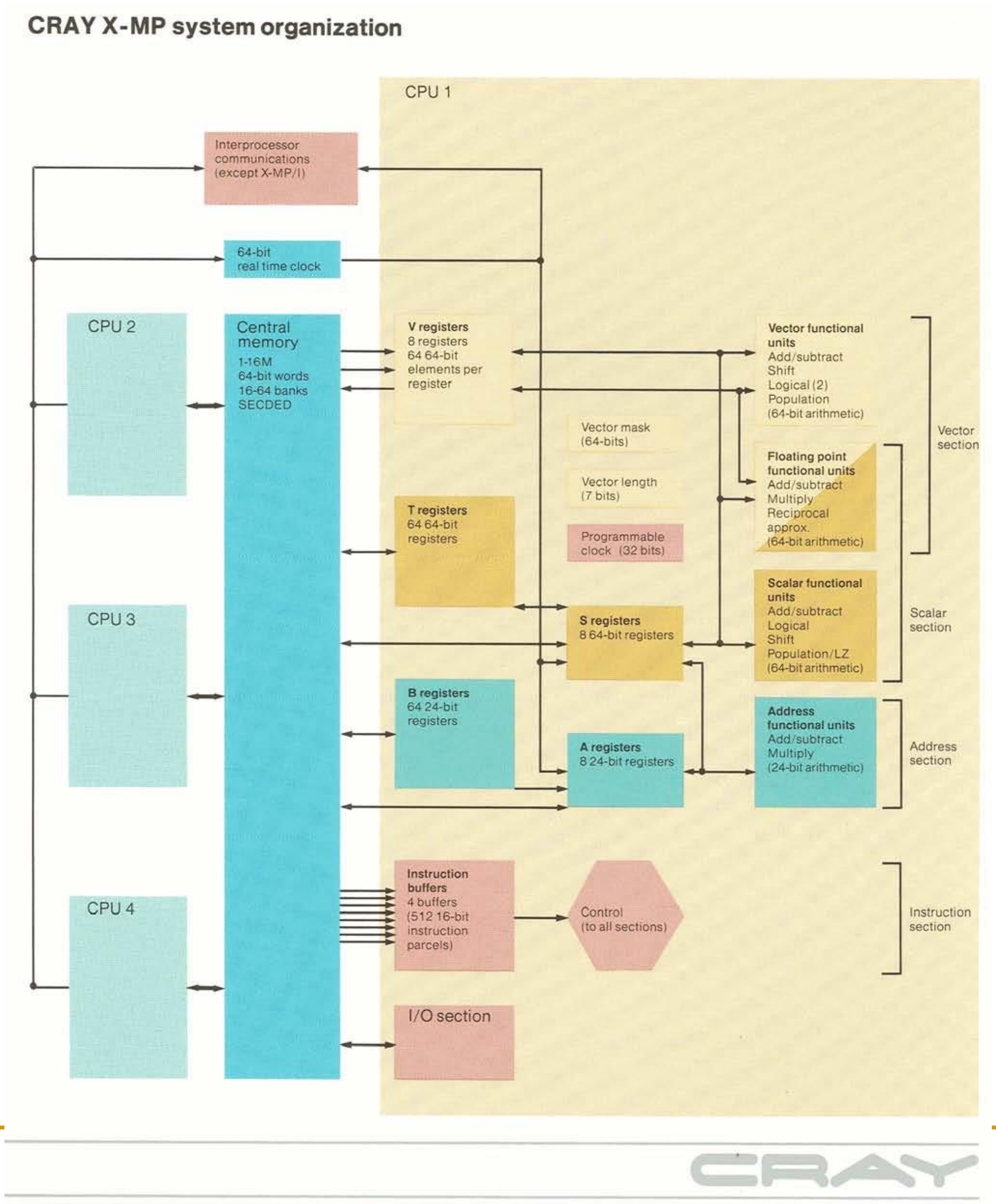
Miniaturisierung und explodierende Leistung

Wie rasend schnell sich die Leistungsfähigkeit der Hardware entwickelt hat, zeigt der Vergleich des gelben Riesen mit einem Minicomputer von heute.

Cray X-MP/28
Zwei parallele Vektorprozessoren mit 118 MHz Systemtakt bringen eine maximale Rechenleistung von 400 Megaflops. Anschaffungspreis 1988: rund 5 Millionen Franken.

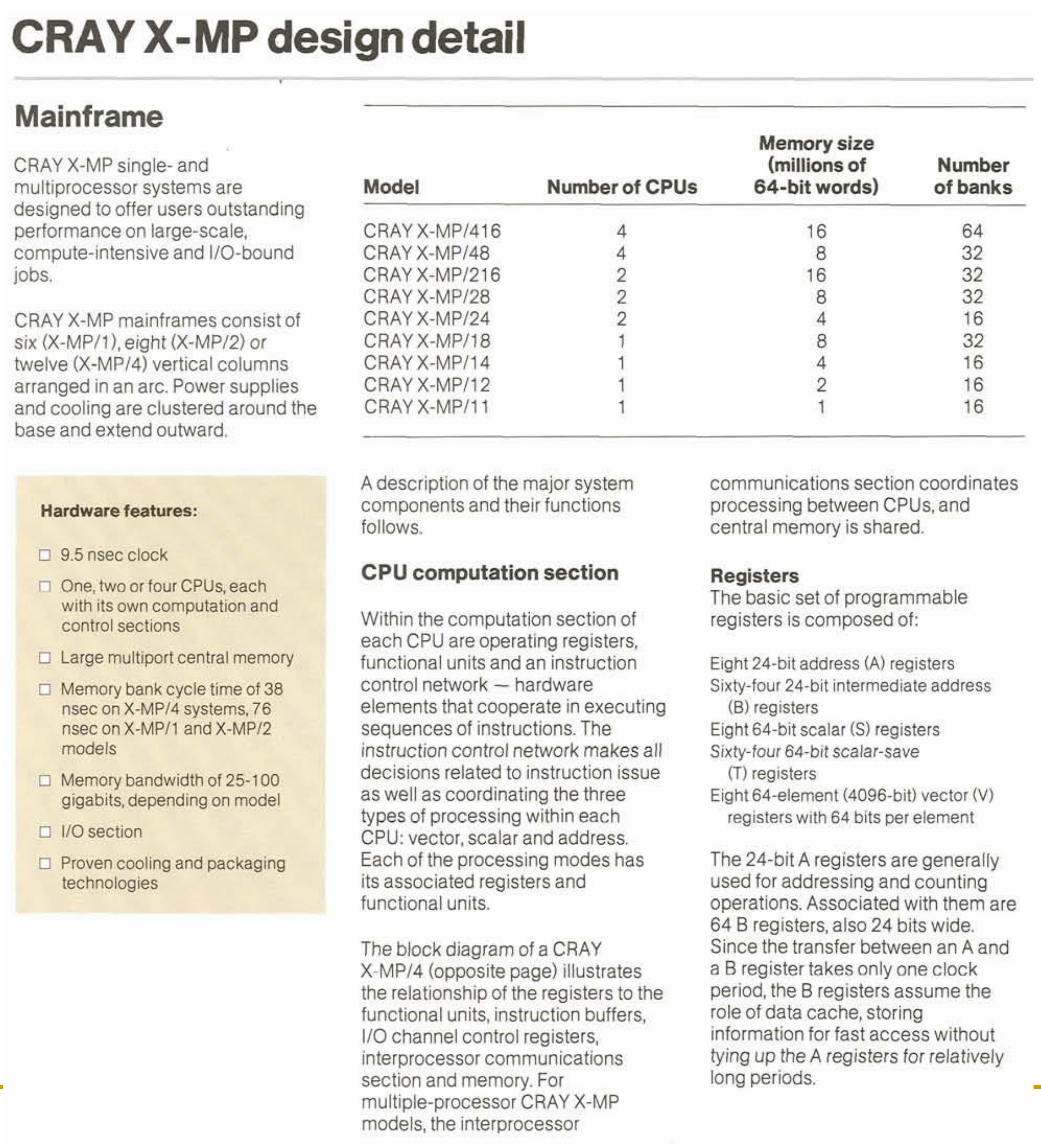
Rhasberry Pi 1 model B+
Die gleichen 400 Megaflops werden heute zum Beispiel von der untenstehenden Einplatinencomputer, ausgestattet mit einem ARM11 Einkernprozessor mit 1000 MHz Systemtakt, erreicht. Anschaffungspreis 2015: 32 Franken.

CRAY X-MP System Organization



Cray Research Inc., “The CRAY X-MP Series of Computer Systems,” 1985

CRAY X-MP Design Detail



CRAY X-MP design detail

Mainframe

CRAY X-MP single- and multiprocessor systems are designed to offer users outstanding performance on large-scale, compute-intensive and I/O-bound jobs.

CRAY X-MP mainframes consist of six (X-MP/1), eight (X-MP/2) or twelve (X-MP/4) vertical columns arranged in an arc. Power supplies and cooling are clustered around the base and extend outward.

Model	Number of CPUs	Memory size (millions of 64-bit words)	Number of banks
CRAY X-MP/416	4	16	64
CRAY X-MP/48	4	8	32
CRAY X-MP/216	2	16	32
CRAY X-MP/28	2	8	32
CRAY X-MP/24	2	4	16
CRAY X-MP/18	1	8	32
CRAY X-MP/14	1	4	16
CRAY X-MP/12	1	2	16
CRAY X-MP/11	1	1	16

Hardware features:

- ☐ 9.5 nsec clock
- ☐ One, two or four CPUs, each with its own computation and control sections
- ☐ Large multiport central memory
- ☐ Memory bank cycle time of 38 nsec on X-MP/4 systems, 76 nsec on X-MP/1 and X-MP/2 models
- ☐ Memory bandwidth of 25-100 gigabits, depending on model
- ☐ I/O section
- ☐ Proven cooling and packaging technologies

A description of the major system components and their functions follows.

CPU computation section

Within the computation section of each CPU are operating registers, functional units and an instruction control network — hardware elements that cooperate in executing sequences of instructions. The instruction control network makes all decisions related to instruction issue as well as coordinating the three types of processing within each CPU: vector, scalar and address. Each of the processing modes has its associated registers and functional units.

The block diagram of a CRAY X-MP/4 (opposite page) illustrates the relationship of the registers to the functional units, instruction buffers, I/O channel control registers, interprocessor communications section and memory. For multiple-processor CRAY X-MP models, the interprocessor communications section coordinates processing between CPUs, and central memory is shared.

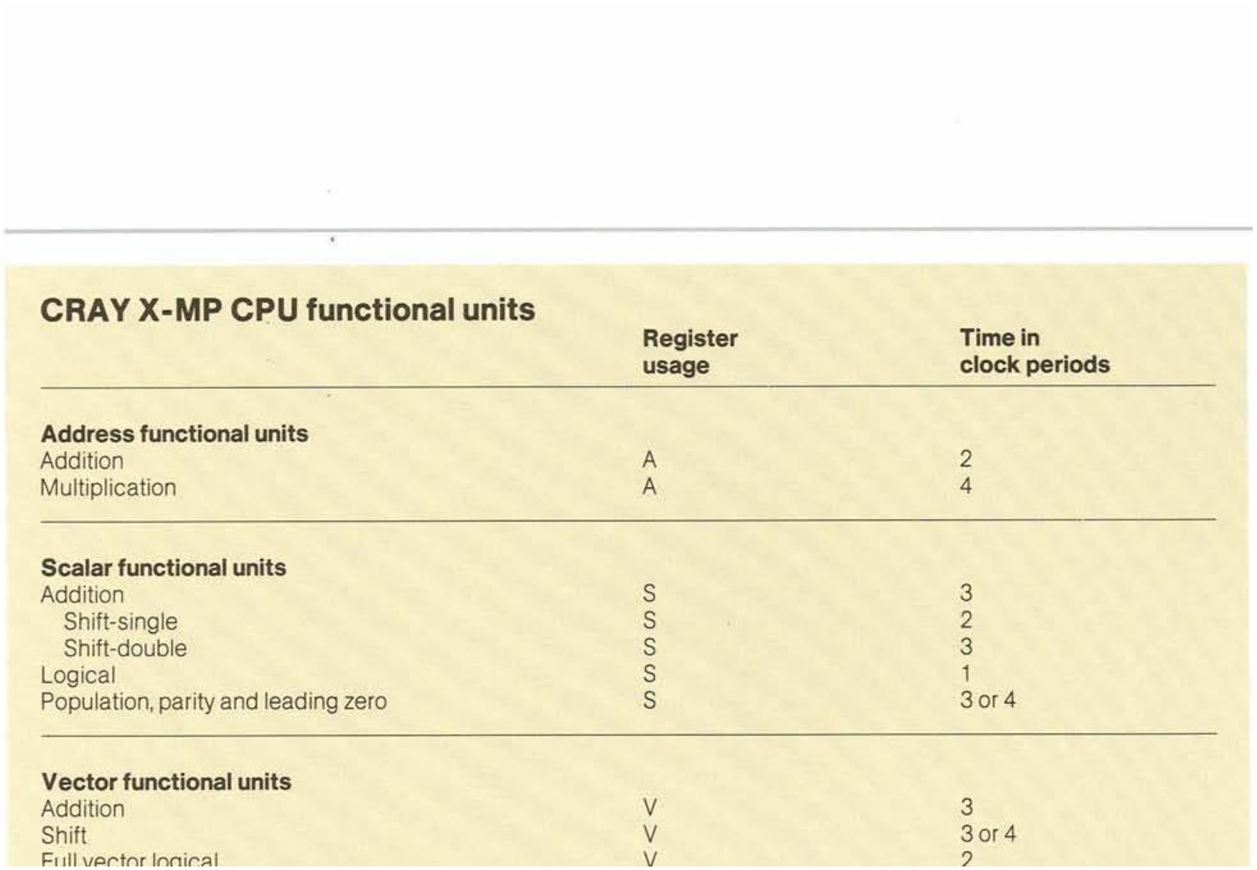
Registers
The basic set of programmable registers is composed of:

Eight 24-bit address (A) registers
Sixty-four 24-bit intermediate address (B) registers
Eight 64-bit scalar (S) registers
Sixty-four 64-bit scalar-save (T) registers
Eight 64-element (4096-bit) vector (V) registers with 64 bits per element

The 24-bit A registers are generally used for addressing and counting operations. Associated with them are 64 B registers, also 24 bits wide. Since the transfer between an A and a B register takes only one clock period, the B registers assume the role of data cache, storing information for fast access without tying up the A registers for relatively long periods.

Cray Research Inc., “The CRAY X-MP Series of Computer Systems,” 1985

CRAY X-MP CPU Functional Units

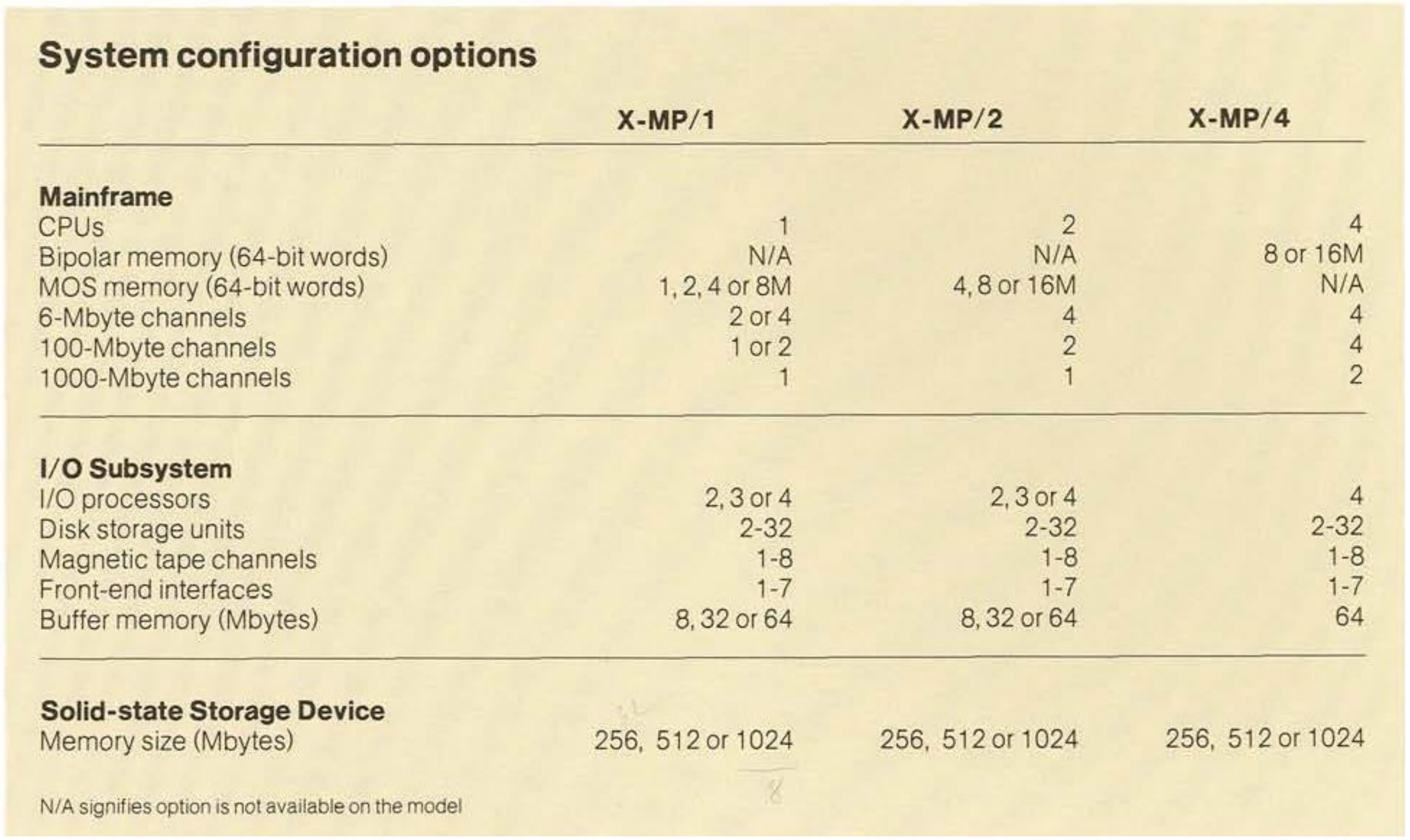


CRAY X-MP CPU functional units

	Register usage	Time in clock periods
Address functional units		
Addition	A	2
Multiplication	A	4
Scalar functional units		
Addition	S	3
Shift-single	S	2
Shift-double	S	3
Logical	S	1
Population, parity and leading zero	S	3 or 4
Vector functional units		
Addition	V	3
Shift	V	3 or 4
Full vector logical	V	2

Cray Research Inc., "The CRAY X-MP Series of Computer Systems," 1985

CRAY X-MP System Configuration



System configuration options

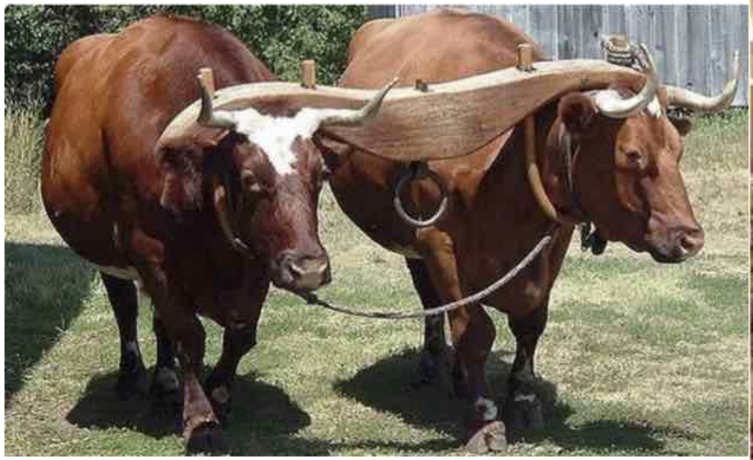
	X-MP/1	X-MP/2	X-MP/4
Mainframe			
CPUs	1	2	4
Bipolar memory (64-bit words)	N/A	N/A	8 or 16M
MOS memory (64-bit words)	1, 2, 4 or 8M	4, 8 or 16M	N/A
6-Mbyte channels	2 or 4	4	4
100-Mbyte channels	1 or 2	2	4
1000-Mbyte channels	1	1	2
I/O Subsystem			
I/O processors	2, 3 or 4	2, 3 or 4	4
Disk storage units	2-32	2-32	2-32
Magnetic tape channels	1-8	1-8	1-8
Front-end interfaces	1-7	1-7	1-7
Buffer memory (Mbytes)	8, 32 or 64	8, 32 or 64	64
Solid-state Storage Device			
Memory size (Mbytes)	256, 512 or 1024	256, 512 or 1024	256, 512 or 1024

N/A signifies option is not available on the model

Cray Research Inc., "The CRAY X-MP Series of Computer Systems," 1985

Seymour Cray, Leader in Supercomputer Design

"If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?"

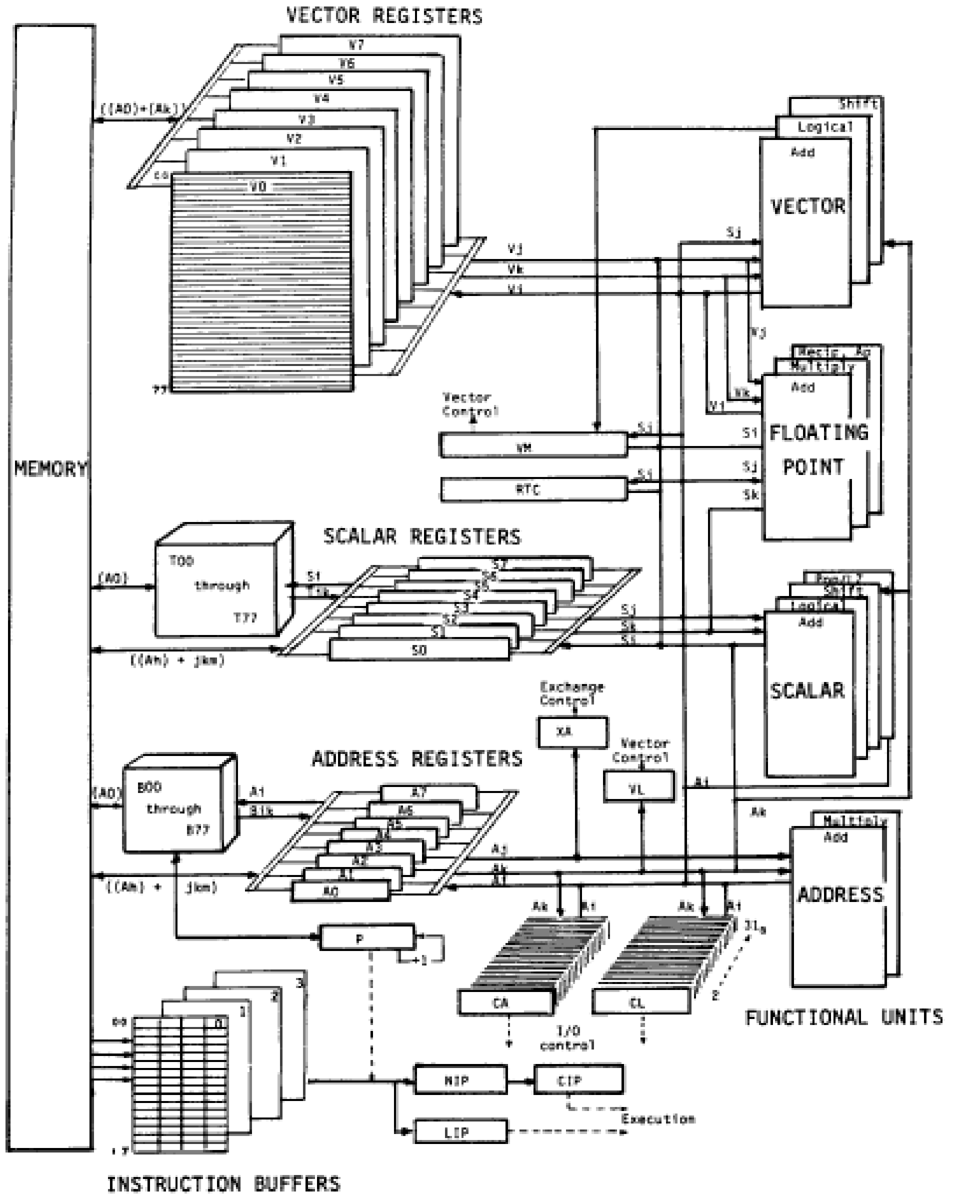


© amityrebecca / Pinterest. https://www.pinterest.ch/pin/473018767088408061/

© Scott Sinklier / Corbis. http://america.aljazeera.com/articles/2015/2/20/the-short-brutal-life-of-male-chickens.html

https://en.wikipedia.org/wiki/Seymour_Cray

Vector Machine Organization (CRAY-1)



- CRAY-1
- Russell, "The CRAY-1 computer system," CACM 1978.

- Scalar and vector modes
- 8 64-element vector registers
- 64 bits per element
- 16 memory banks
- 8 64-bit scalar registers
- 8 24-bit address registers

Loading/Storing Vectors from/to Memory

- Requires loading/storing multiple elements

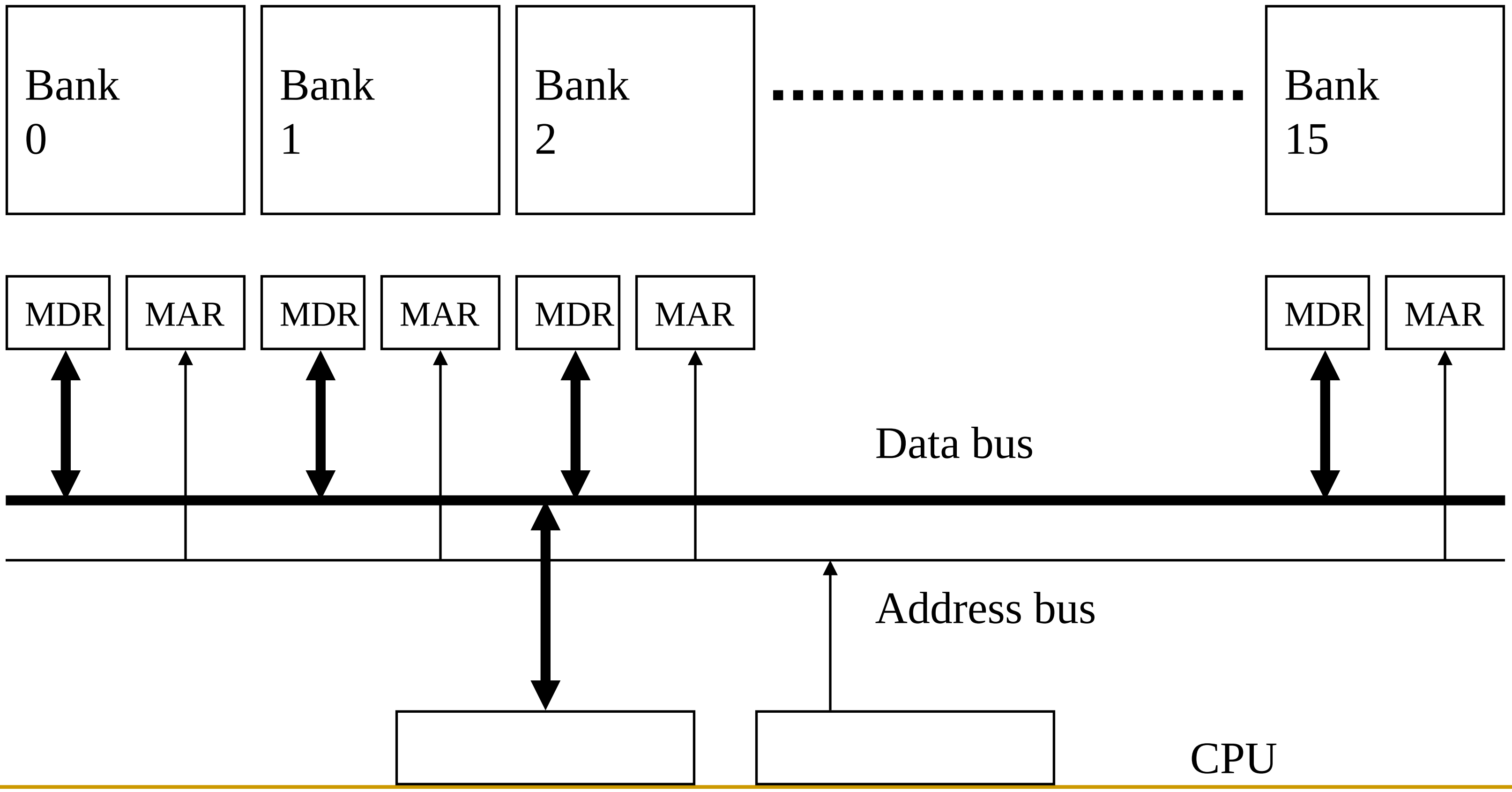
- Elements separated from each other by a constant distance (stride)
  - Assume stride = 1 for now

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

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

Memory Banking

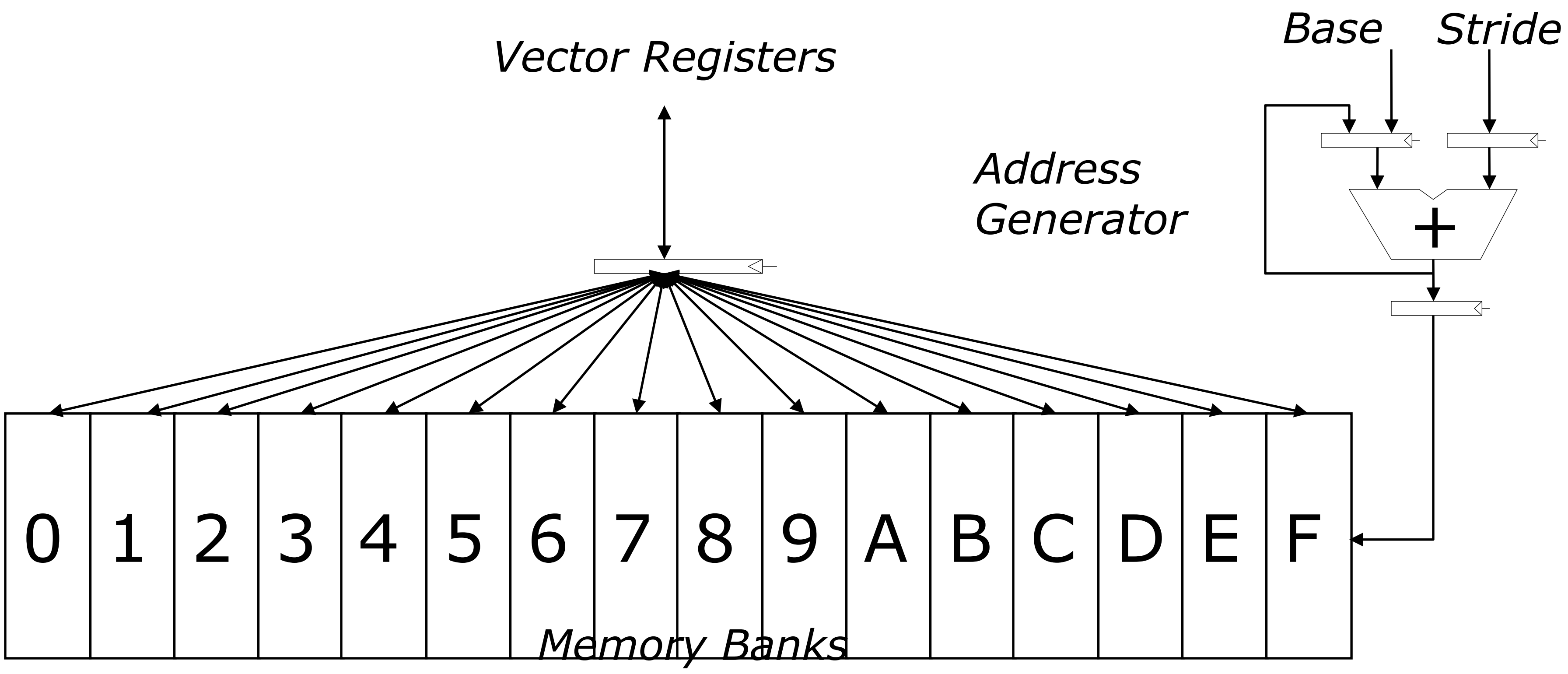
- Memory is divided into banks that can be accessed independently; banks share address and data buses (to minimize pin cost)
- Can start and complete one bank access per cycle
- Can sustain N concurrent accesses if all N go to different banks



Picture credit: Derek Chiou

Vector Memory System

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



Picture credit: Krste Asanovic

Scalar Code Example: Element-Wise Avg.

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

- Scalar code (instruction and its latency)

```
       MOVI R0 = 50                  1
       MOVA R1 = A                   1          304 dynamic instructions
       MOVA R2 = B                   1
       MOVA R3 = C                   1
X:     LD R4 = MEM[R1++]             11  ;autoincrement addressing
       LD R5 = MEM[R2++]             11
       ADD R6 = R4 + R5              4
       SHFR R7 = R6 >> 1             1
       ST MEM[R3++] = R7             11
       DECBNZ R0, X                  2   ;decrement and branch if NZ
```

Scalar Code Execution Time (In Order)

- Scalar execution time on an in-order processor with 1 bank
  - First two loads in the loop cannot be pipelined: 2*11 cycles
  - 4 + 50*40 = 2004 cycles

- Scalar execution time on an in-order processor with 16 banks (word-interleaved: consecutive words are stored in consecutive banks)
  - First two loads in the loop can be pipelined
  - 4 + 50*30 = 1504 cycles

- Why 16 banks?
  - 11-cycle memory access latency
  - Having 16 (>11) banks ensures there are enough banks to overlap enough memory operations to cover memory latency

Vectorizable Loops

- A loop is vectorizable if each iteration is independent of any other

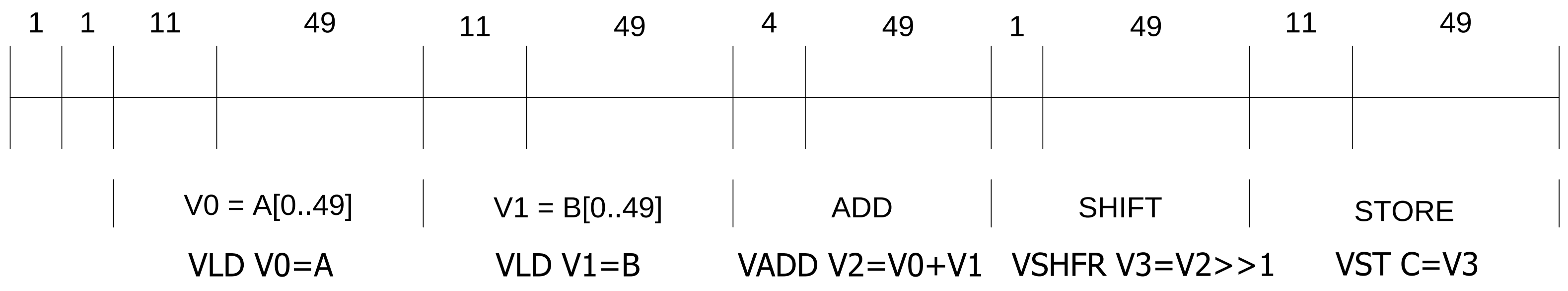
- For I = 0 to 49
  - C[i] = (A[i] + B[i]) / 2
- Vectorized loop (each instruction and its latency):

MOVI VLEN = 50	1
MOVI VSTR = 1	1
VLD V0 = A	11 + VLEN – 1
VLD V1 = B	11 + VLEN – 1
VADD V2 = V0 + V1	4 + VLEN – 1
VSHFR V3 = V2 >> 1	1 + VLEN – 1
VST C = V3	11 + VLEN – 1

7 dynamic instructions

Basic Vector Code Performance

- Assume no chaining (no vector data forwarding)
  - i.e., output of a vector functional unit cannot be used as the direct input of another
  - The entire vector register needs to be ready before any element of it can be used as part of another operation
- One memory port (one address generator)
- 16 memory banks (word-interleaved)

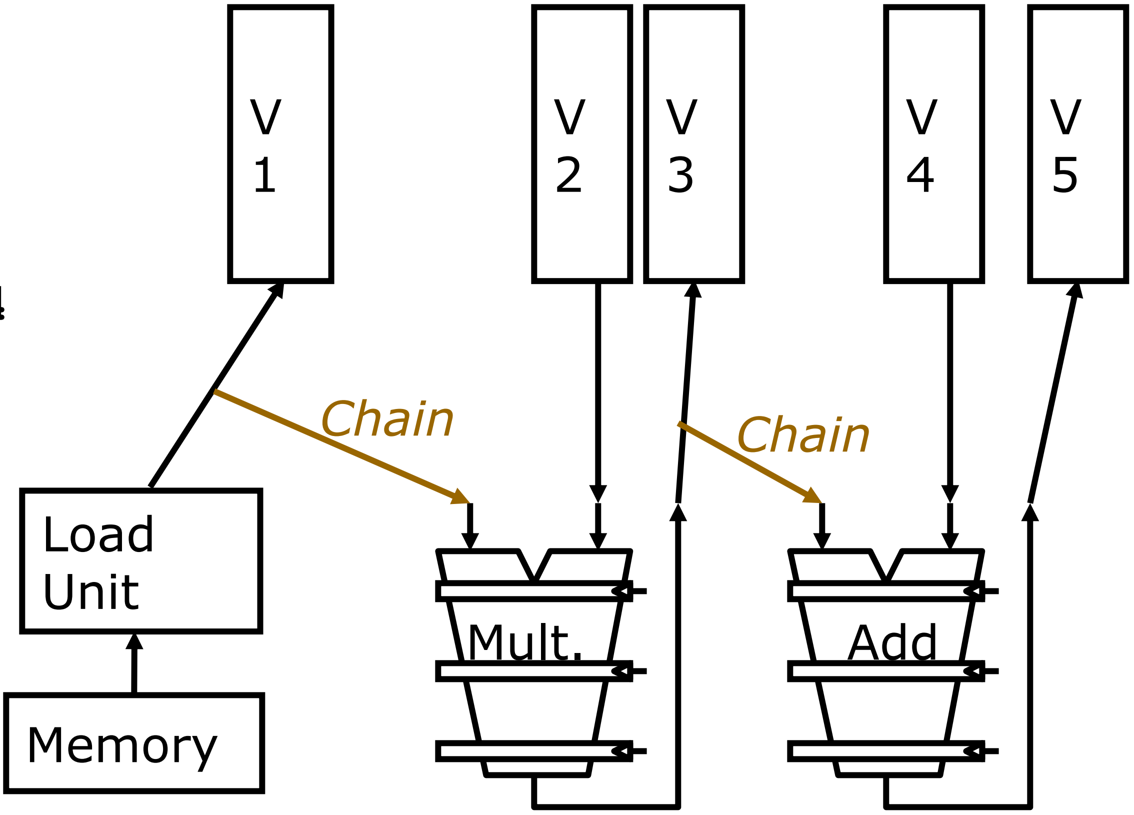


- 285 cycles

Vector Chaining

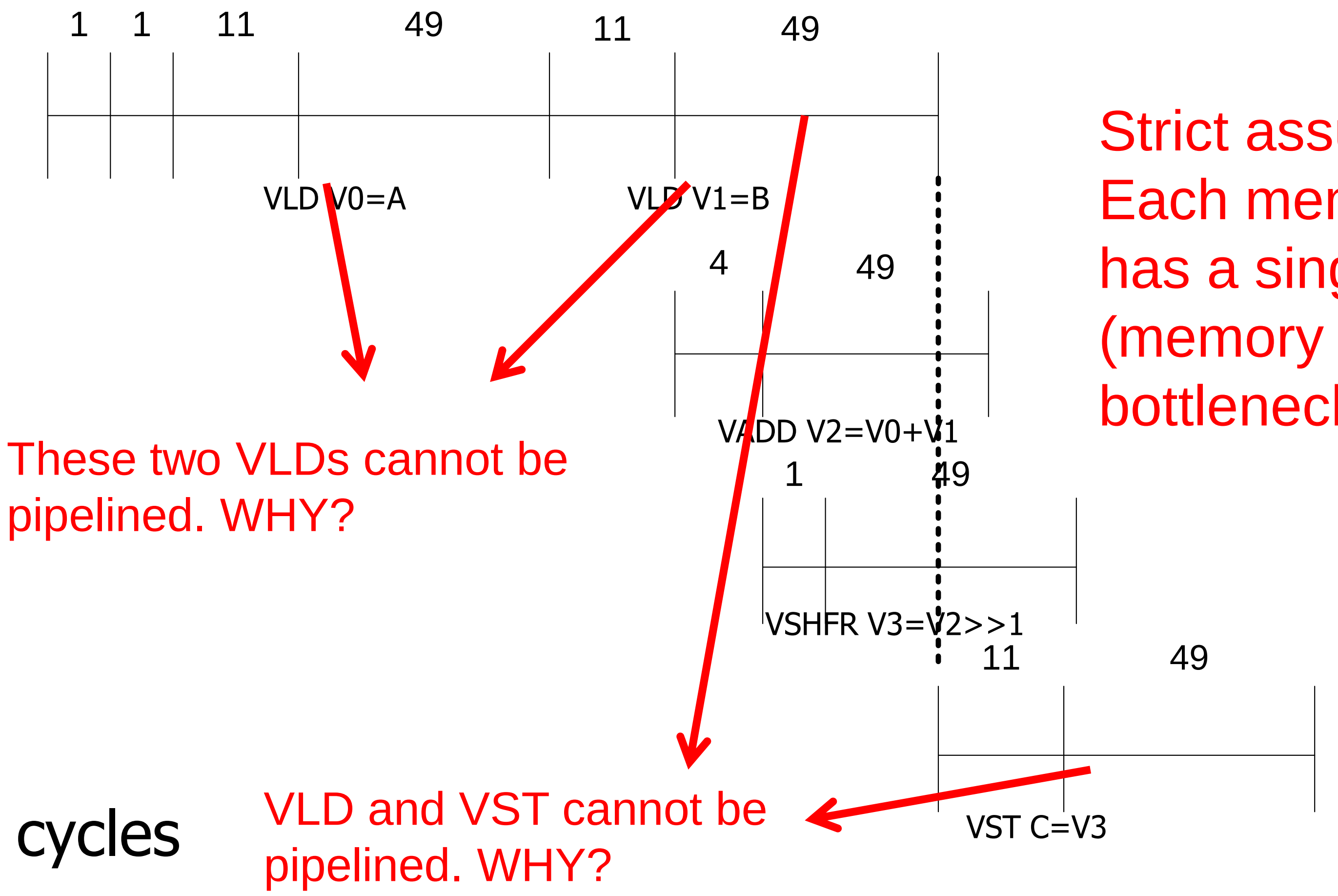
- Vector chaining: Data forwarding from one vector functional unit to another

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



Vector Code Performance - Chaining

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

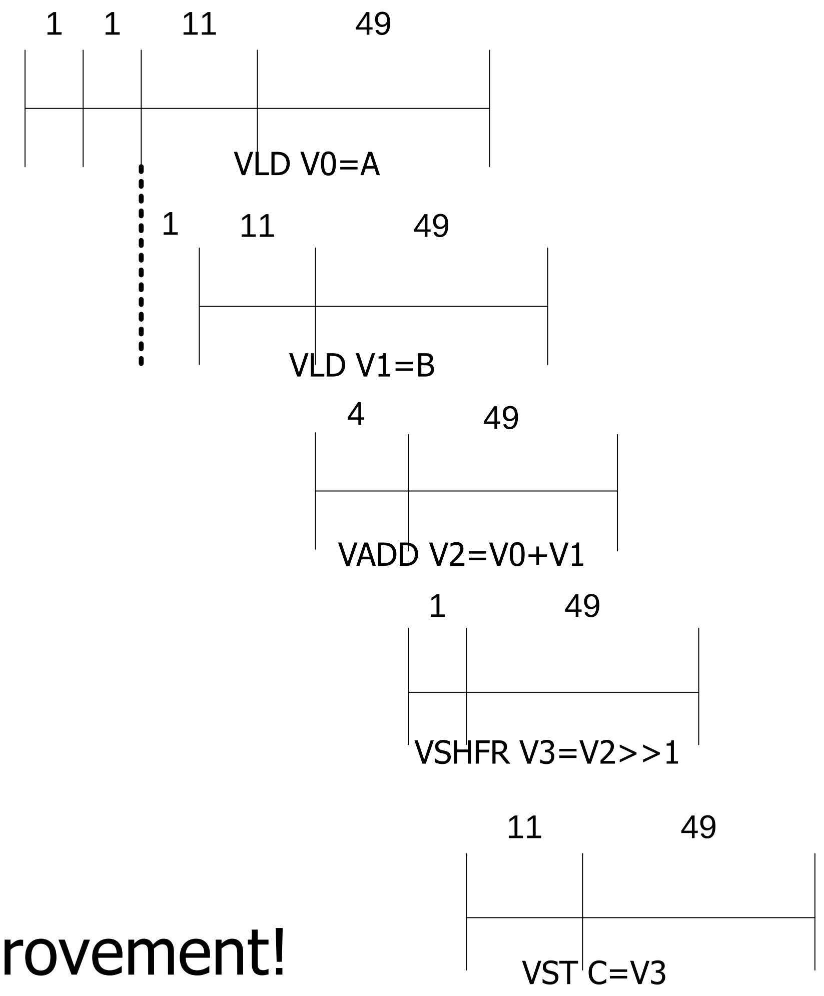

1
1
11
49
11
49
VLD V0=A
VLD V1=B
4
49
VADD V2=V0+V1
1
49
VSHFR V3=V2>>1
11
49
VST C=V3
These two VLDs cannot be pipelined. WHY?
VLD and VST cannot be pipelined. WHY?

Strict assumption: Each memory bank has a single port (memory bandwidth bottleneck)

- 182 cycles

Vector Code Performance – Multiple Memory Ports

- Chaining and 2 load ports, 1 store port in each bank



- 79 cycles
- 19X perf. improvement!

Questions (I)

- What if # data elements > # elements in a vector register?
  - Idea: Break loops so that each iteration operates on # elements in a vector register
    - E.g., 527 data elements, 64-element VREGs
    - 8 iterations where VLEN = 64
    - 1 iteration where VLEN = 15 (need to change value of VLEN)
  - Called vector stripmining

(Vector) Stripmining

Surface mining, including **strip mining**, open-pit mining and mountaintop removal mining, is a broad category of mining in which soil and rock overlying the mineral deposit (the overburden) are removed, in contrast to underground mining, in which the overlying rock is left in place, and the mineral removed through shafts or tunnels.

Surface mining began in the mid-sixteenth century[1] and is practiced throughout the world, although the majority of surface coal mining occurs in North America.[2] It gained

Coal strip mine in Wyoming

Source: https://en.wikipedia.org/wiki/Surface_mining

Questions (II)

- What if vector data is not stored in a strided fashion in memory? (irregular memory access to a vector)
  - Idea: Use indirection to combine/pack elements into vector registers
  - Called scatter/gather operations
  - Doing so also helps with avoiding useless computation on sparse vectors (i.e., vectors where many elements are 0)

Gather/Scatter Operations

Want to vectorize loops with indirect accesses:

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

Indexed load instruction (*Gather*)

```
LV vD, rD          # Load indices in D vector
LVI vC, rC, vD     # Load indirect from rC base
LV vB, rB          # Load B vector
ADDV.D vA,vB,vC    # Do add
SV vA, rA          # Store result
```

Gather/Scatter Operations

- Gather/scatter operations often implemented in hardware to handle sparse vectors (matrices) or indirect indexing
- Vector loads and stores use an index vector which is added to the base register to generate the addresses
- *Scatter* example

Index Vector	Data Vector (to Store)	Stored Vector (in Memory)	
0	3.14	Base+0	3.14
2	6.5	Base+1	X
6	71.2	Base+2	6.5
7	2.71	Base+3	X
		Base+4	X
		Base+5	X
		Base+6	71.2
		Base+7	2.71

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]
```

A	B	VMASK
1	2	0
2	2	1
3	2	1
4	10	0
-5	-4	0
0	-3	1
6	5	1
-7	-8	1

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

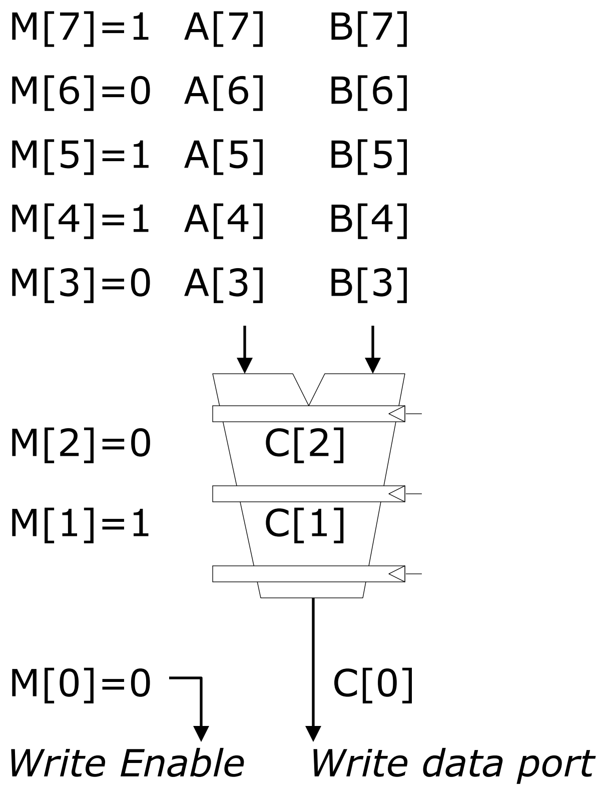
Masked Vector Instructions

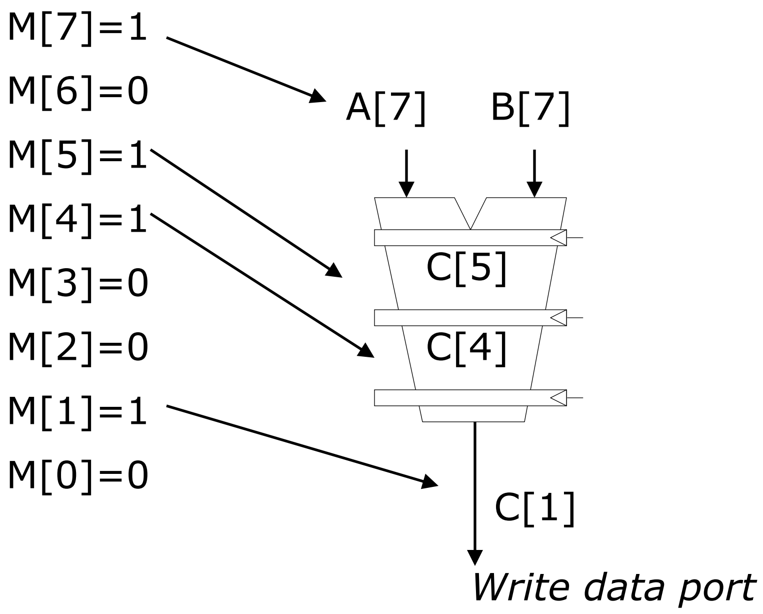
Simple Implementation

– execute all N operations, turn off result writeback according to mask

Density-Time Implementation

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





Which one is better?

Tradeoffs?

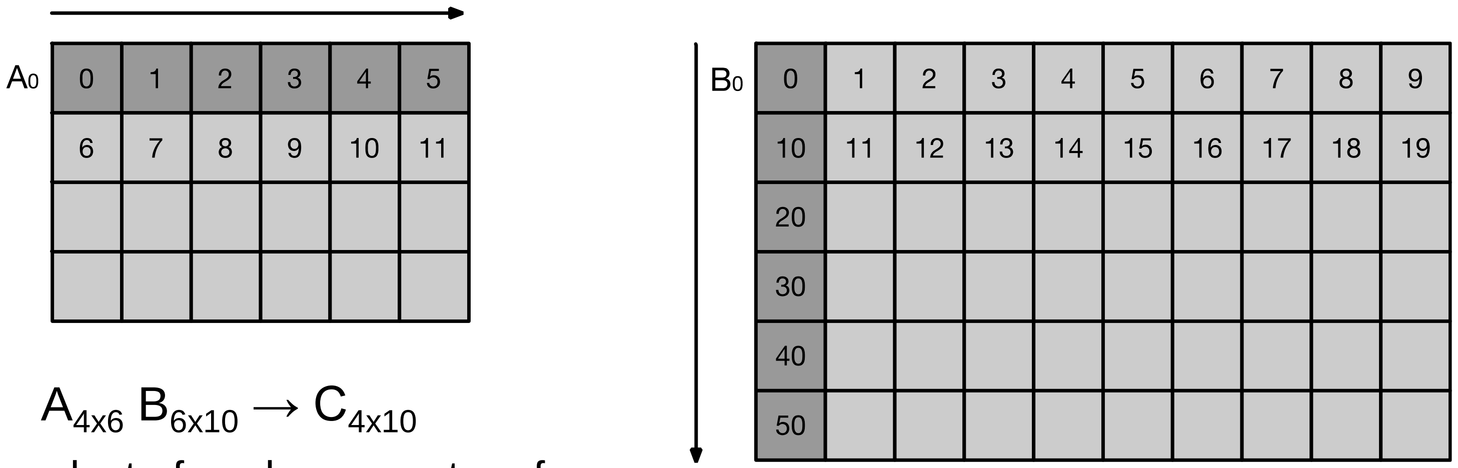
Some Issues

- 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

Bank Conflicts in Matrix Multiplication

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



$A_{4x6}\ B_{6x10} \rightarrow C_{4x10}$

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

- Load A's row 0 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 into vector register V_2
  - Each time, increment address by 10
  - Accesses have a stride of 10

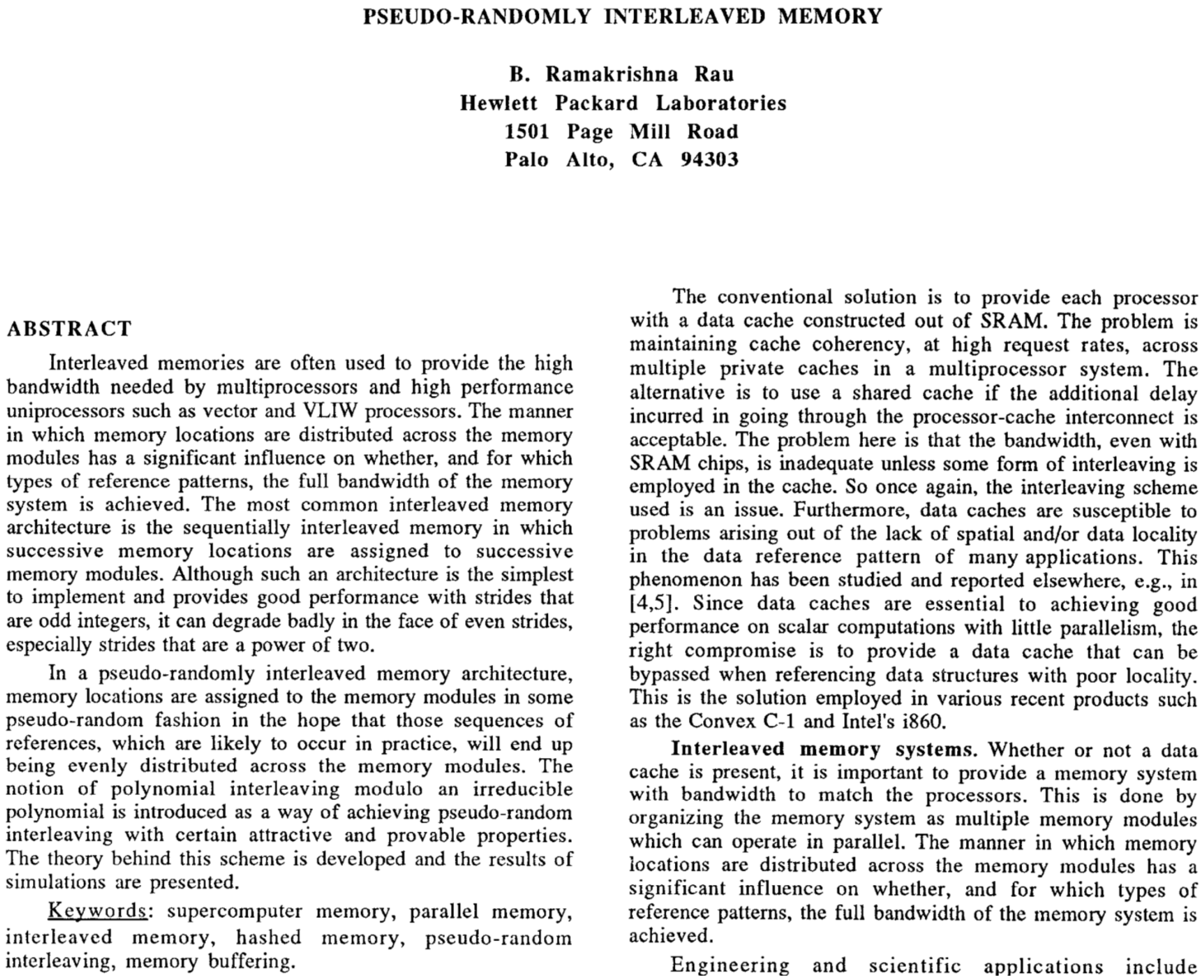
Different strides can lead to bank conflicts

How do we minimize them?

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
  - Rau, "Pseudo-randomly interleaved memory," ISCA 1991.

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

Rau, "Pseudo-randomly Interleaved Memory," ISCA 1991.

Array vs. Vector Processors, Revisited

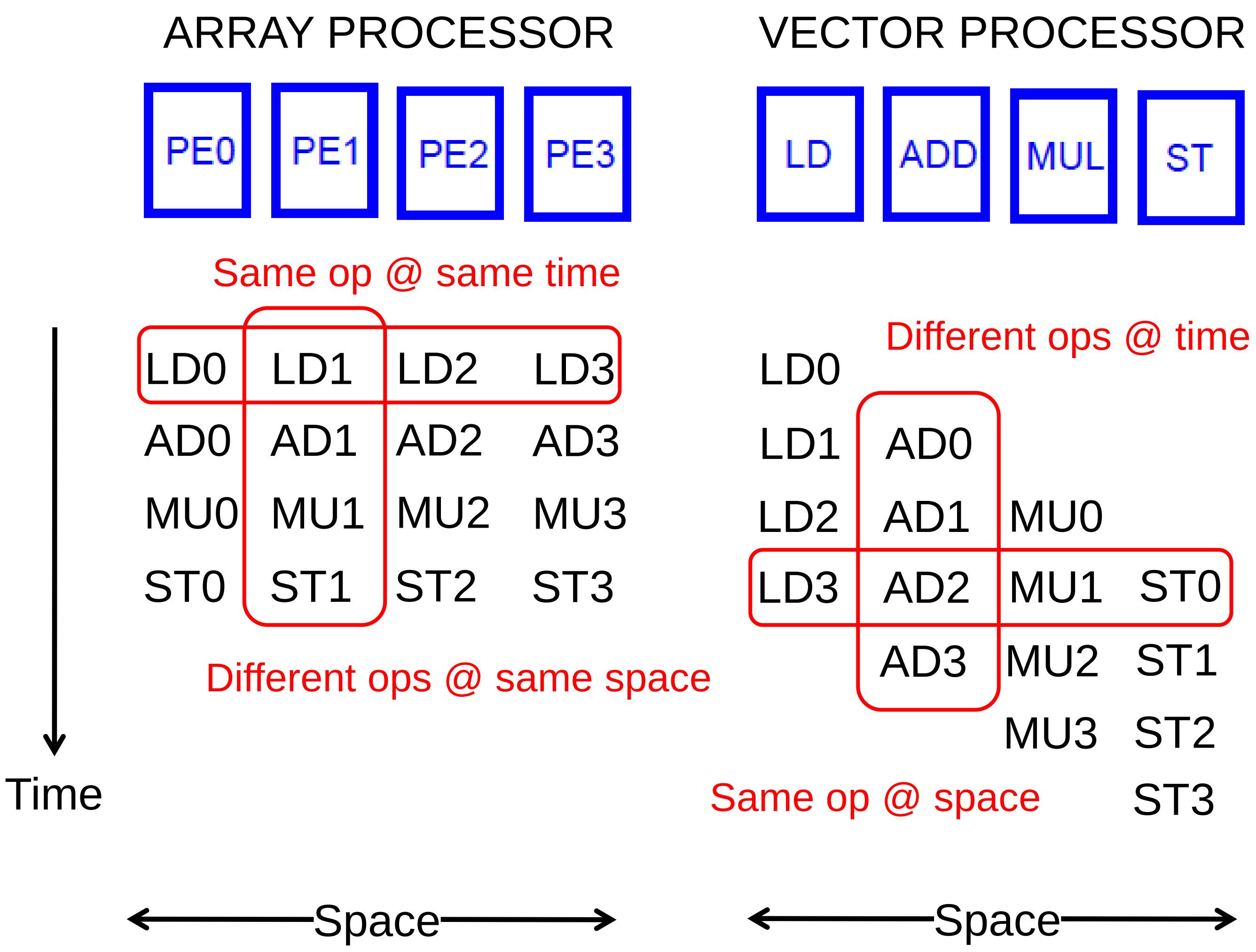
- Array vs. vector processor distinction is a “purist’s” distinction

- Most “modern” SIMD processors are a combination of both
  - They exploit data parallelism in both time and space
  - GPUs are a prime example we will cover in a bit more detail

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



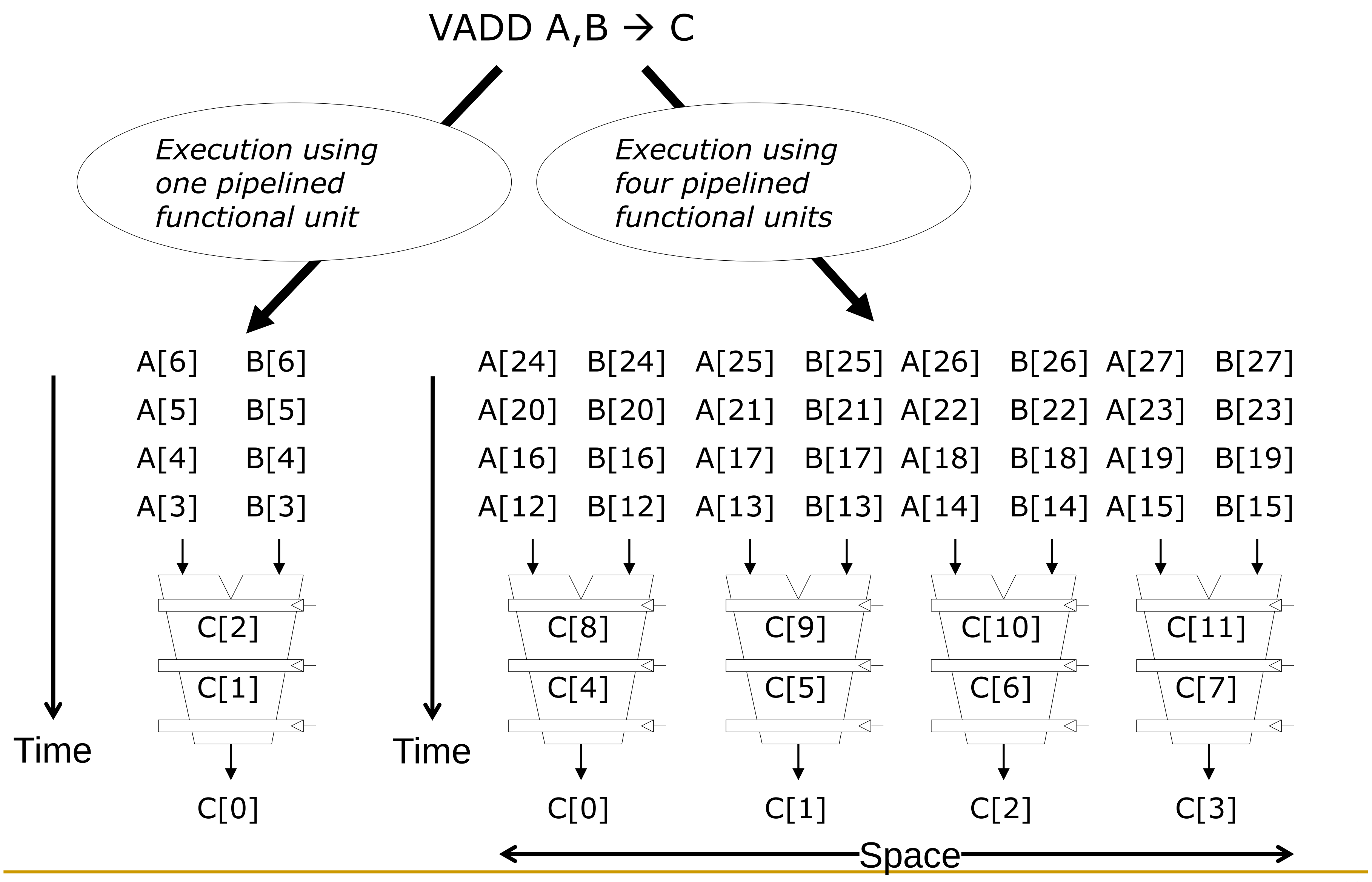
ARRAY PROCESSOR

PE0	PE1	PE2	PE3
LD0	LD1	LD2	LD3
AD0	AD1	AD2	AD3
MU0	MU1	MU2	MU3
ST0	ST1	ST2	ST3

VECTOR PROCESSOR

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

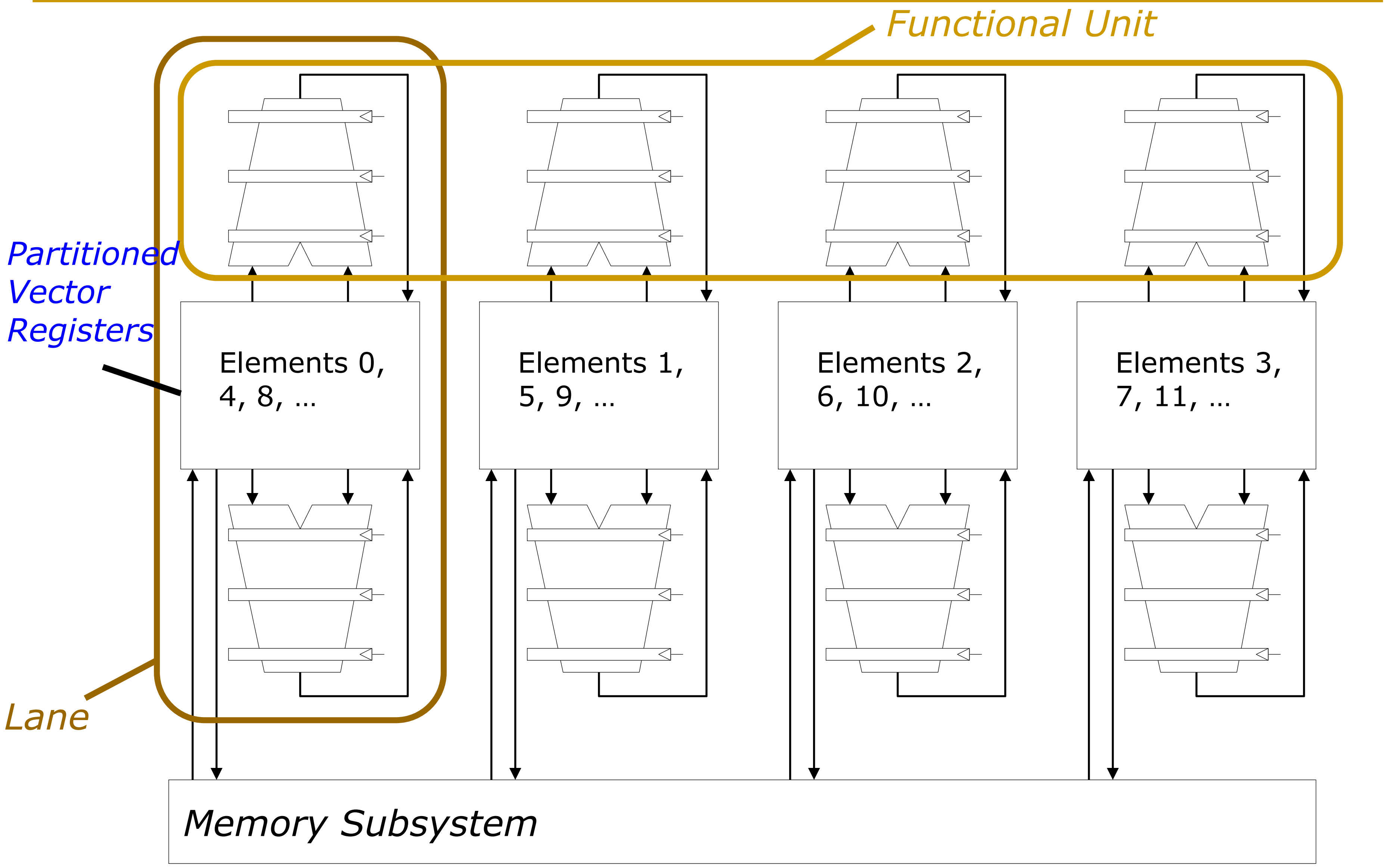
Vector Instruction Execution



VADD A,B → C
Execution using one pipelined functional unit
Execution using four pipelined functional units
A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]
C[2]
C[1]
C[0]
Time
A[24] B[24] A[25] B[25] A[26] B[26] A[27] B[27]
A[20] B[20] A[21] B[21] A[22] B[22] A[23] B[23]
A[16] B[16] A[17] B[17] A[18] B[18] A[19] B[19]
A[12] B[12] A[13] B[13] A[14] B[14] A[15] B[15]
C[8] C[9] C[10] C[11]
C[4] C[5] C[6] C[7]
C[0] C[1] C[2] C[3]
Time
Space

Slide credit: Krste Asanovic

Vector Unit Structure

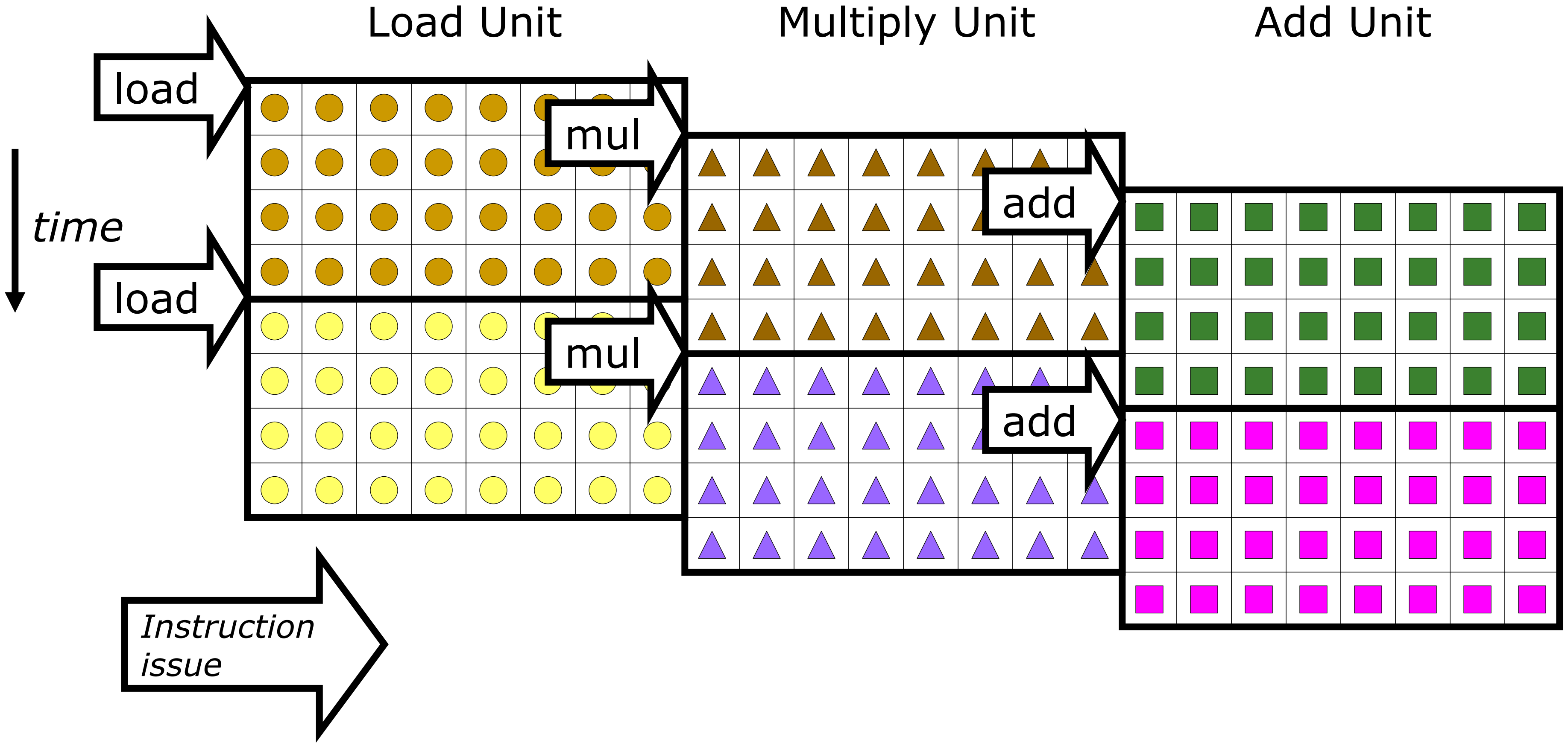


Slide credit: Krste Asanovic

Vector Instruction Level Parallelism

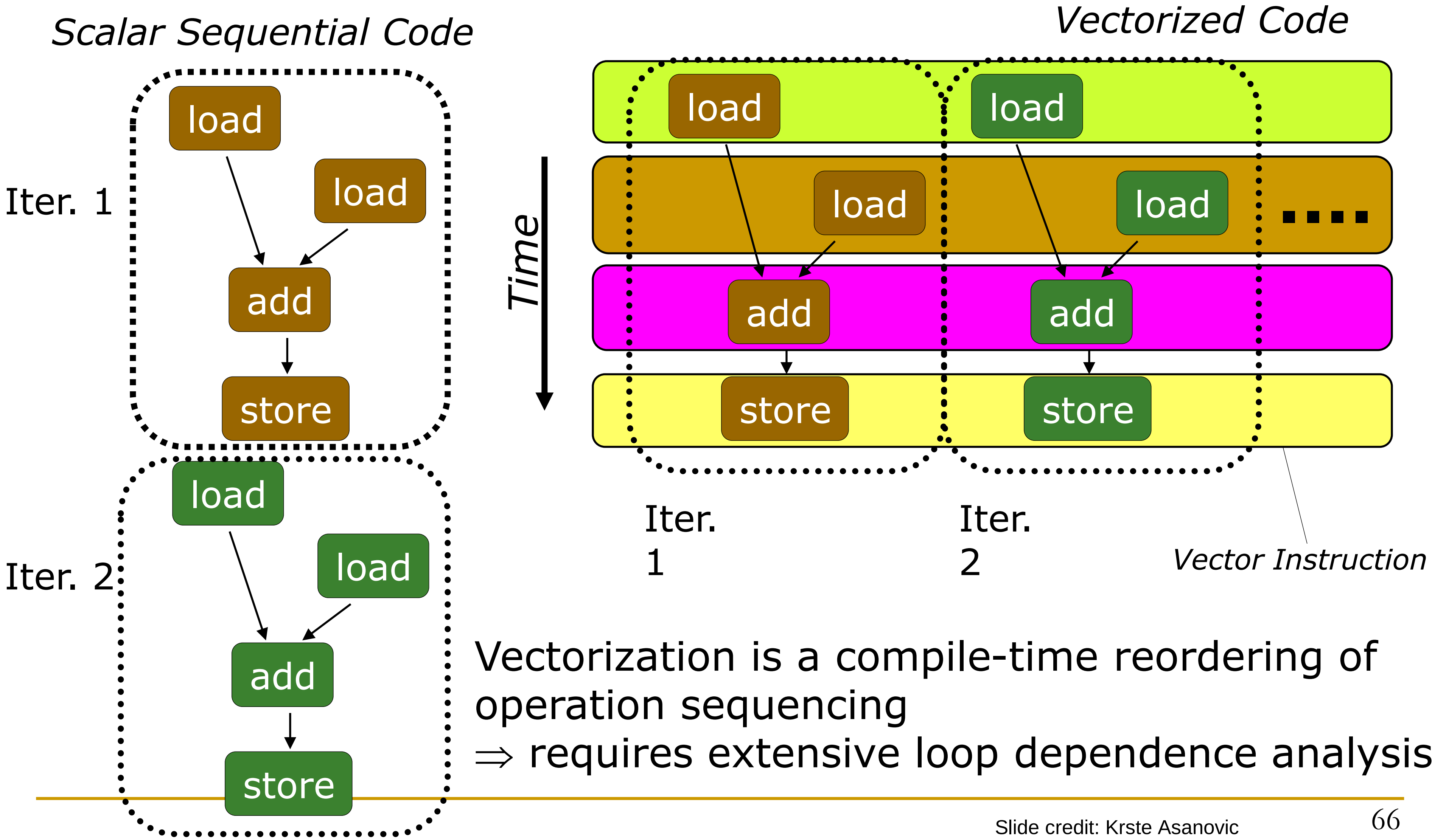
Can overlap execution of multiple vector instructions

- Example machine has 32 elements per vector register and 8 lanes
- Completes 24 operations/cycle while issuing 1 vector instruction/cycle



Automatic Code Vectorization

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



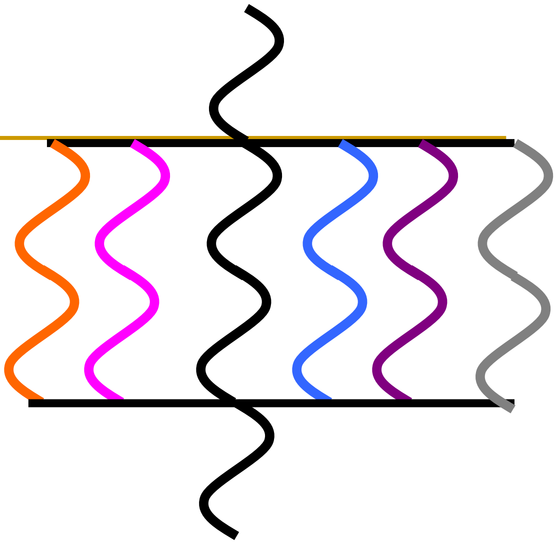
Vectorization is a compile-time reordering of operation sequencing
⇒ requires extensive loop dependence analysis

Slide credit: Krste Asanovic

Vector/SIMD Processing Summary

- Vector/SIMD machines are good at exploiting regular data-level parallelism
  - Same operation performed on many data elements
  - Improve performance, simplify design (no intra-vector dependencies)
- Performance improvement limited by vectorizability of code
  - Scalar operations limit vector machine performance
  - Remember Amdahl's Law
  - CRAY-1 was the fastest SCALAR machine at its time!
- Many existing ISAs include (vector-like) SIMD operations
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD

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

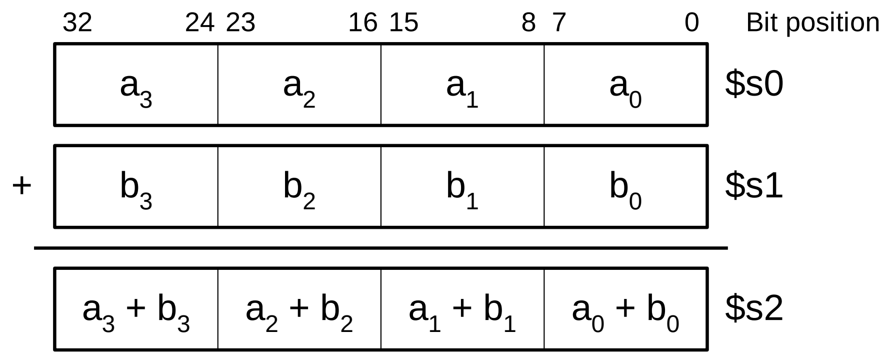
- All parallel machines "suffer from" the serial bottleneck

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

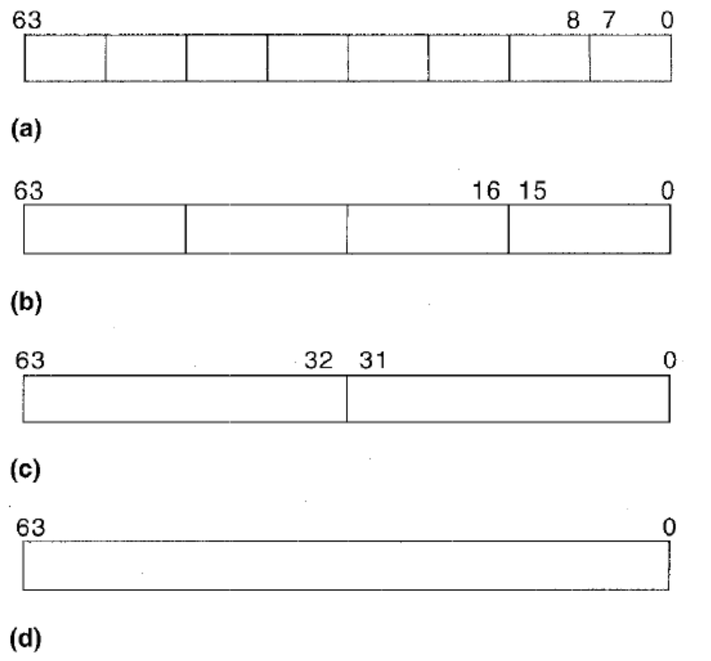


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

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

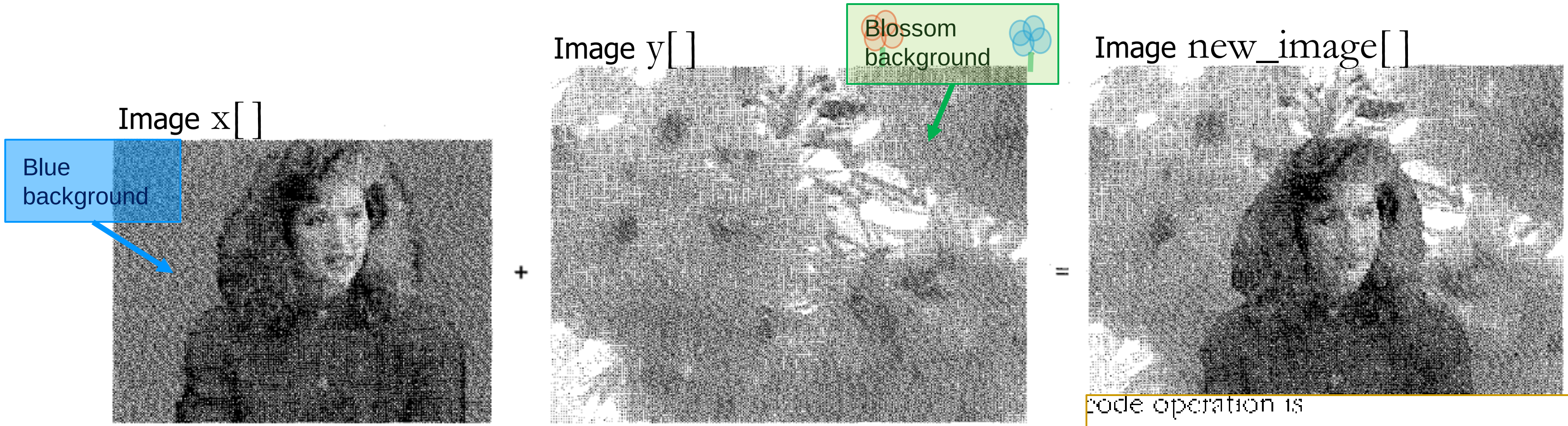


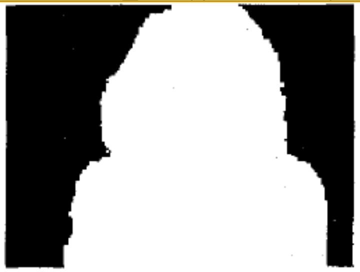
Figure 8. Chroma keying: image overlay using a background color.

```
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];
```

PCMPEQB MM1, MM3

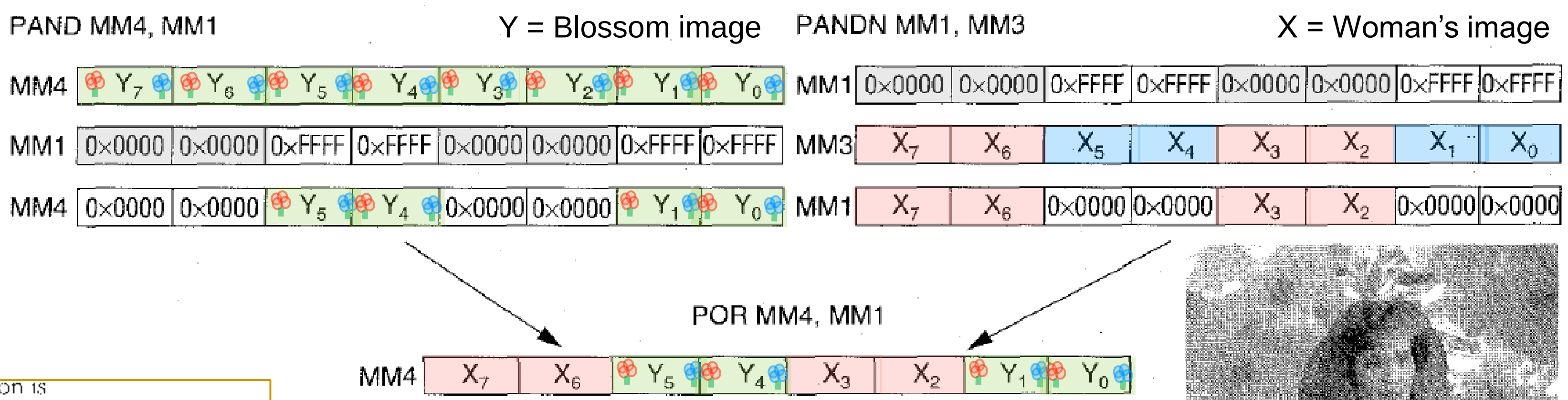
	MM1	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue
Image x[ ]	MM3	X7!=blue	X6!=blue	X5=blue	X4=blue	X3!=blue	X2!=blue	X1=blue	X0=blue
Bit mask	MM1	0x0000	0x0000	0xFFFF	0xFFFF	0x0000	0x0000	0xFFFF	0xFFFF

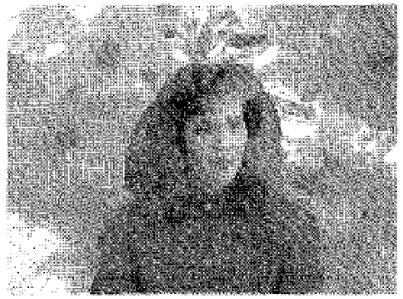


Bitmask

Figure 9. Generating the selection bit mask.

MMX Example: Image Overlaying (II)


PAND MM4, MM1
Y = Blossom image
PANDN MM1, MM3
X = Woman's image
MM4 | Y7 | Y6 | Y5 | Y4 | Y3 | Y2 | Y1 | Y0
MM1 | 0×0000 | 0×0000 | 0×FFFF | 0×FFFF | 0×0000 | 0×0000 | 0×FFFF | 0×FFFF
MM4 | 0×0000 | 0×0000 | Y5 | Y4 | 0×0000 | 0×0000 | Y1 | Y0
MM1 | 0×0000 | 0×0000 | 0×FFFF | 0×FFFF | 0×0000 | 0×0000 | 0×FFFF | 0×FFFF
MM3 | X7 | X6 | X5 | X4 | X3 | X2 | X1 | X0
MM1 | X7 | X6 | 0×0000 | 0×0000 | X3 | X2 | 0×0000 | 0×0000
POR MM4, MM1
MM4 | X7 | X6 | Y5 | Y4 | X3 | X2 | Y1 | Y0



```
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];
```

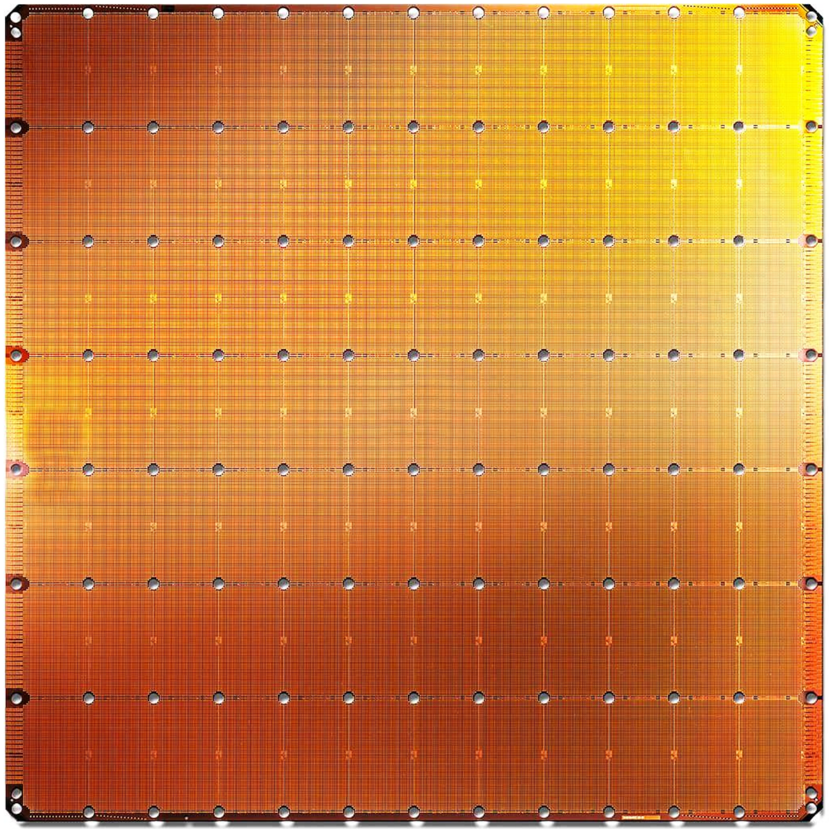
Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

```
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.

SIMD Operations in Modern (Machine Learning) Accelerators

Cerebras's Wafer Scale Engine (2019)



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

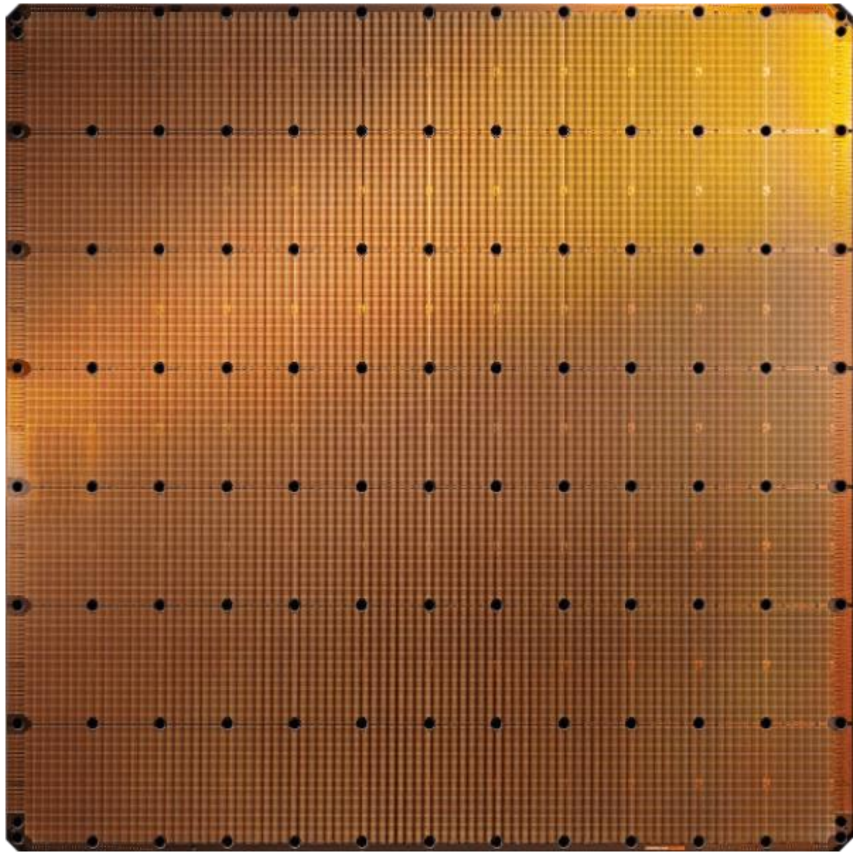
Cerebras WSE
1.2 Trillion transistors
46,225 mm^2

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

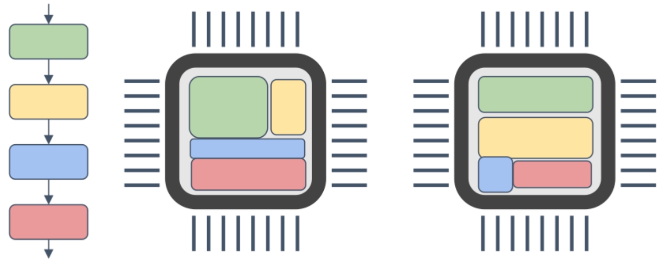
https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning

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

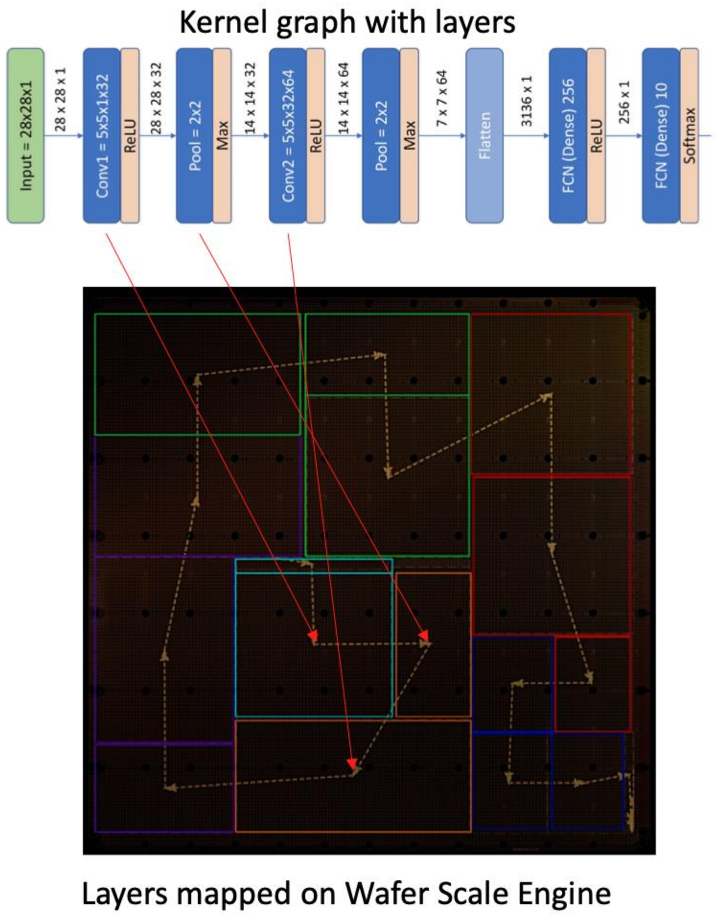
Size, Place, and Route in Cerebras's WSE

- Neural network mapping onto the whole wafer is a challenge

Multiple possible mappings



An example mapping



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."

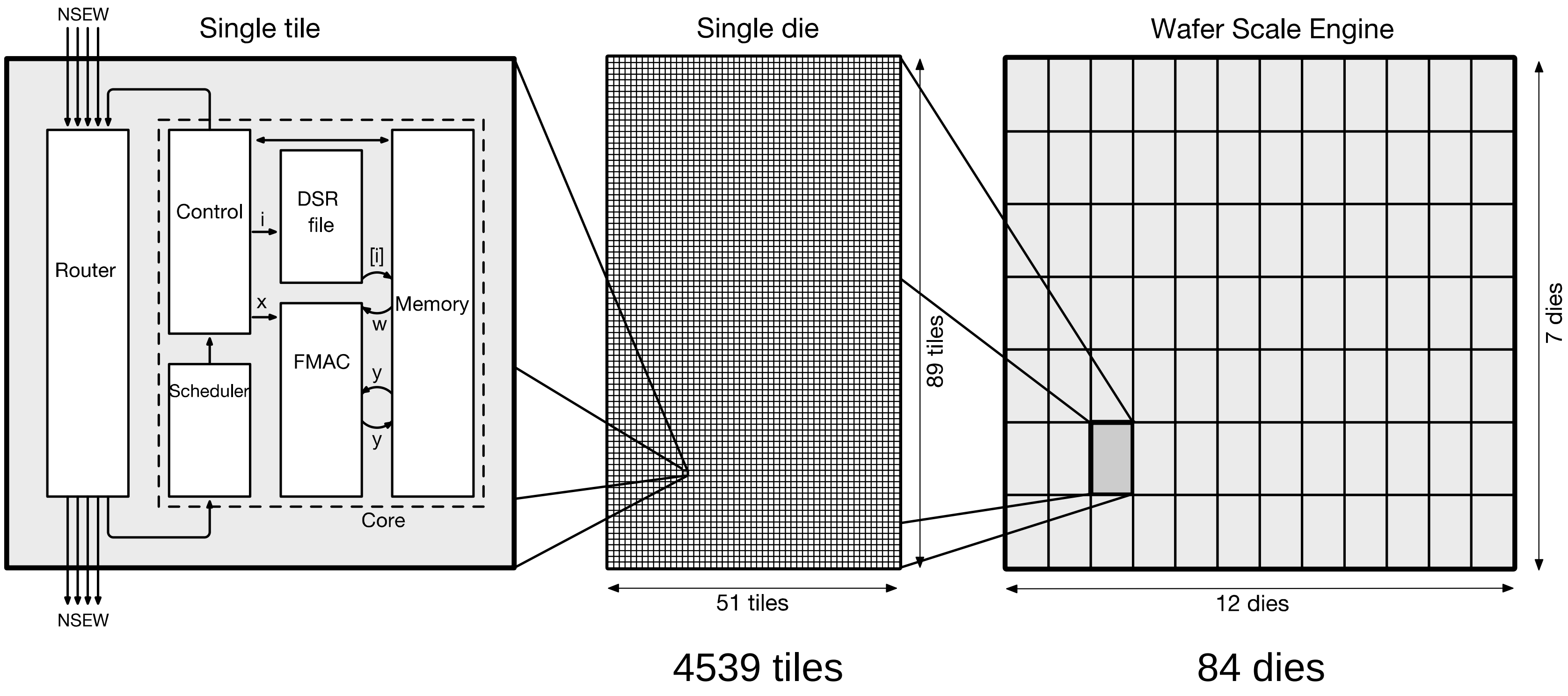
Recall: Flynn's Taxonomy of Computers

- 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

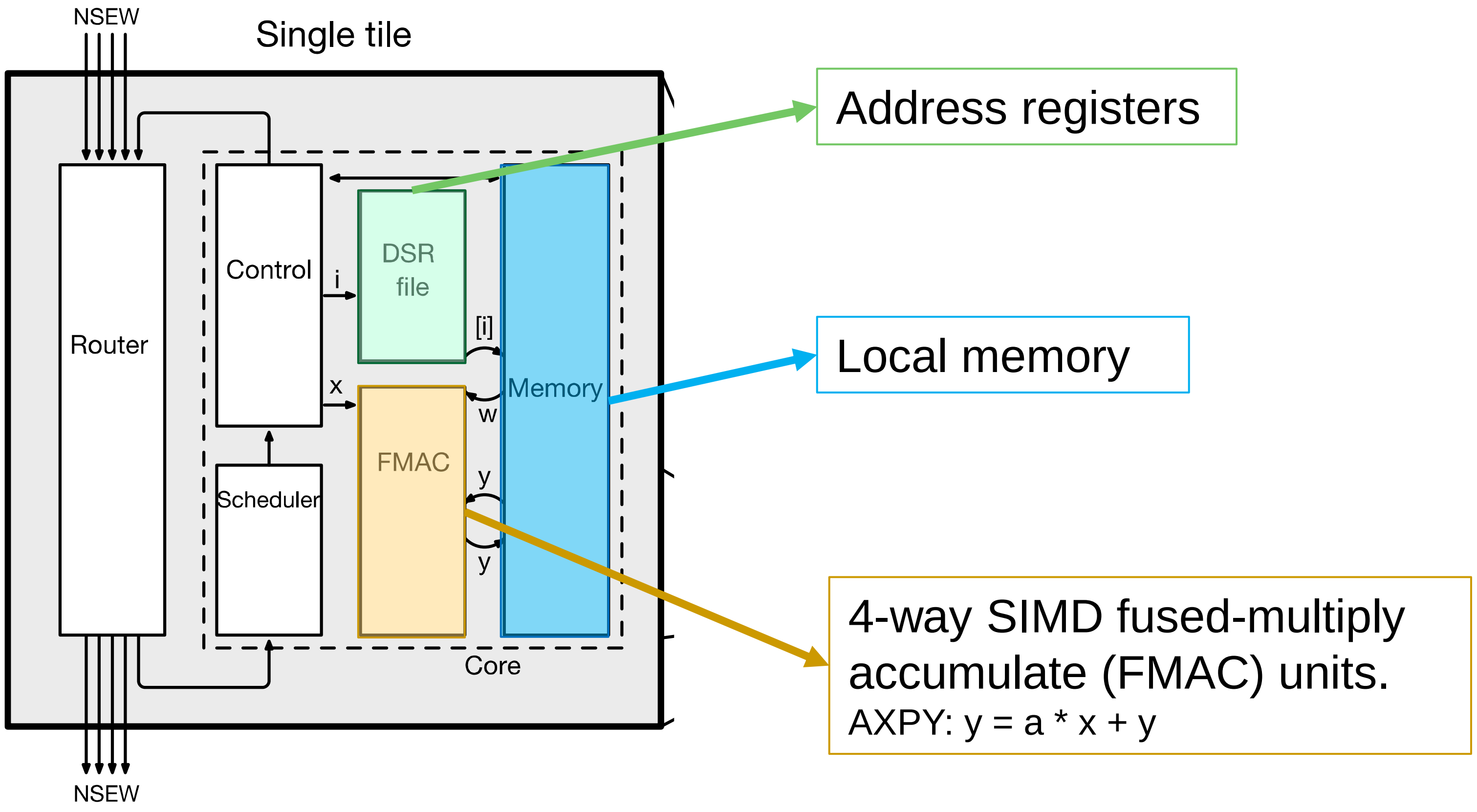
A MIMD Machine with SIMD Processors (I)

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



A MIMD Machine with SIMD Processors (II)

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



Fine-Grained Multithreading

Fine-Grained Multithreading

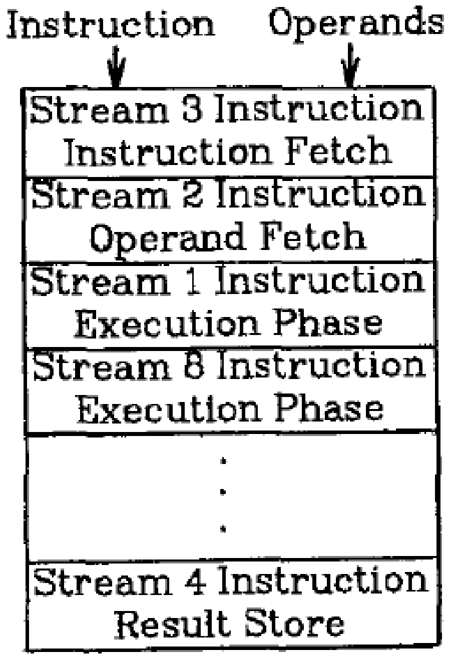
- Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
  - Branch/instruction resolution latency overlapped with execution of other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



Instruction
Operands
Stream 3 Instruction Instruction Fetch
Stream 2 Instruction Operand Fetch
Stream 1 Instruction Execution Phase
Stream 8 Instruction Execution Phase
Stream 4 Instruction Result Store

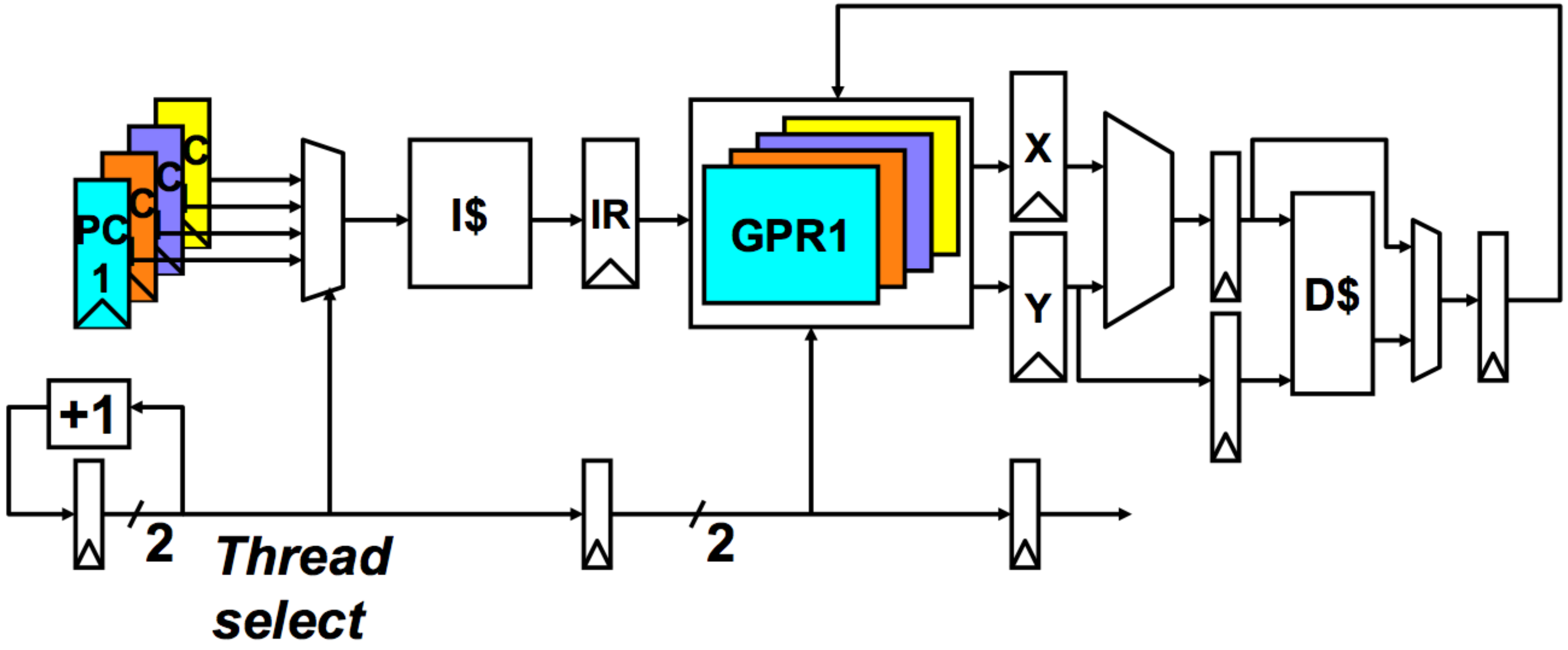
Fine-Grained Multithreading (II)

- 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

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

Multithreaded Pipeline Example



Slide credit: Joel Emer

Fine-grained Multithreading (III)

- Advantages

  + No need for dependency 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, utilization

- Disadvantages

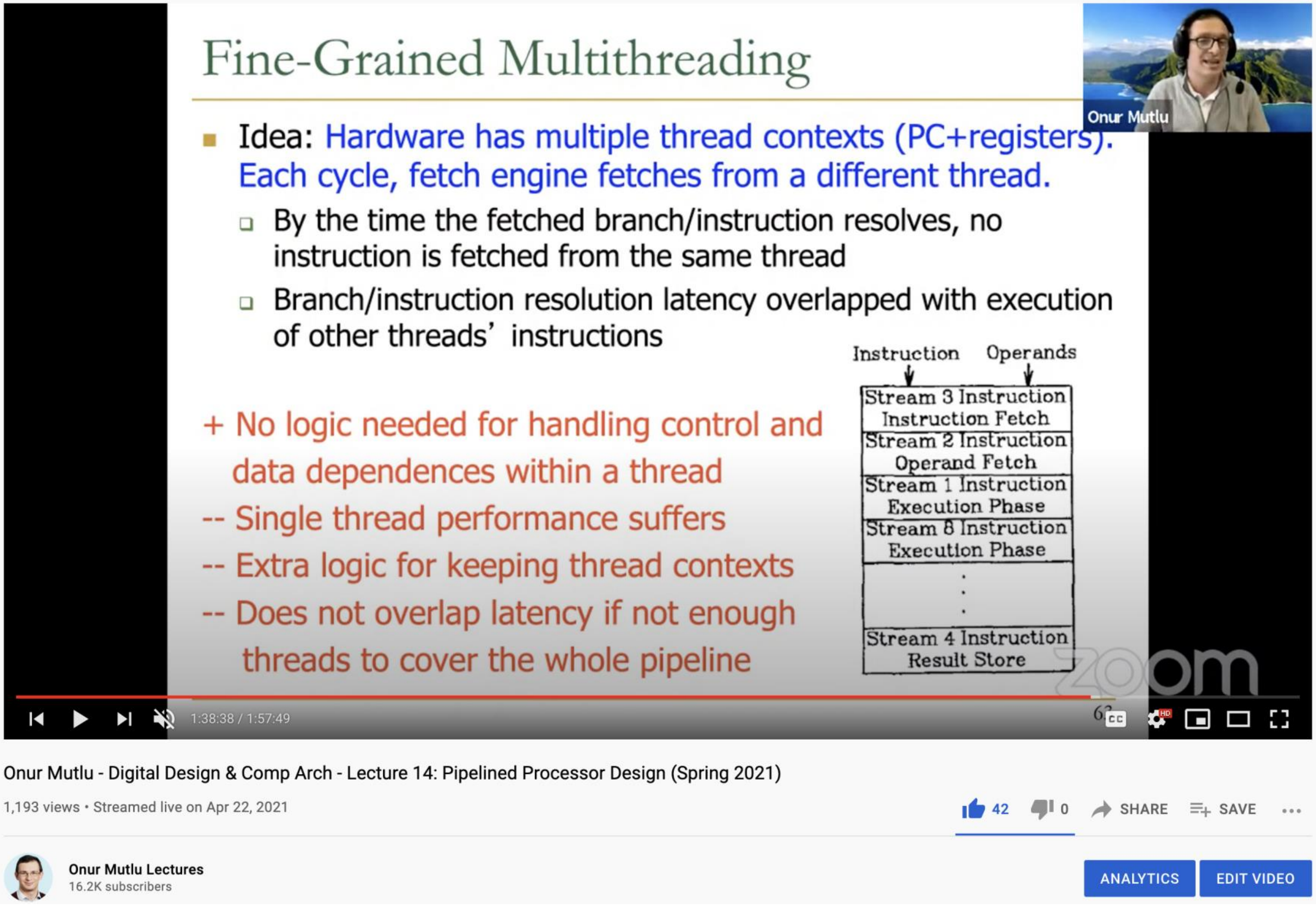
  - Extra hardware complexity: multiple hardware contexts (PCs, register files, …), thread selection logic

  - Reduced single thread performance (one instruction fetched every N cycles from the same thread)

  - Resource contention between threads in caches and memory

  - Some dependency checking logic *between* threads remains (load/store)

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

Instruction
Operands
Stream 3 Instruction Instruction Fetch
Stream 2 Instruction Operand Fetch
Stream 1 Instruction Execution Phase
Stream 8 Instruction Execution Phase
Stream 4 Instruction Result Store
Onur Mutlu
zoom
1:38:38 / 1:57:49
42
0
SHARE
SAVE
ANALYTICS
EDIT VIDEO

Onur Mutlu - Digital Design & Comp Arch - Lecture 14: Pipelined Processor Design (Spring 2021)

1,193 views • Streamed live on Apr 22, 2021

Onur Mutlu Lectures
16.2K subscribers

https://www.youtube.com/watch?v=6e5KZcCGBYw&list=PL5Q2soXY2Zi_uej3aY39YB5pfW4SJ7LlN&index=16

Lectures on Fine-Grained Multithreading

- Digital Design & Computer Architecture, Spring 2021, Lecture 14
  - Pipelined Processor Design (ETH, Spring 2021)
  - https://www.youtube.com/watch?v=6e5KZcCGBYw&list=PL5Q2soXY2Zi_uej3aY39YB5pfW4SJ7LlN&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

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), …

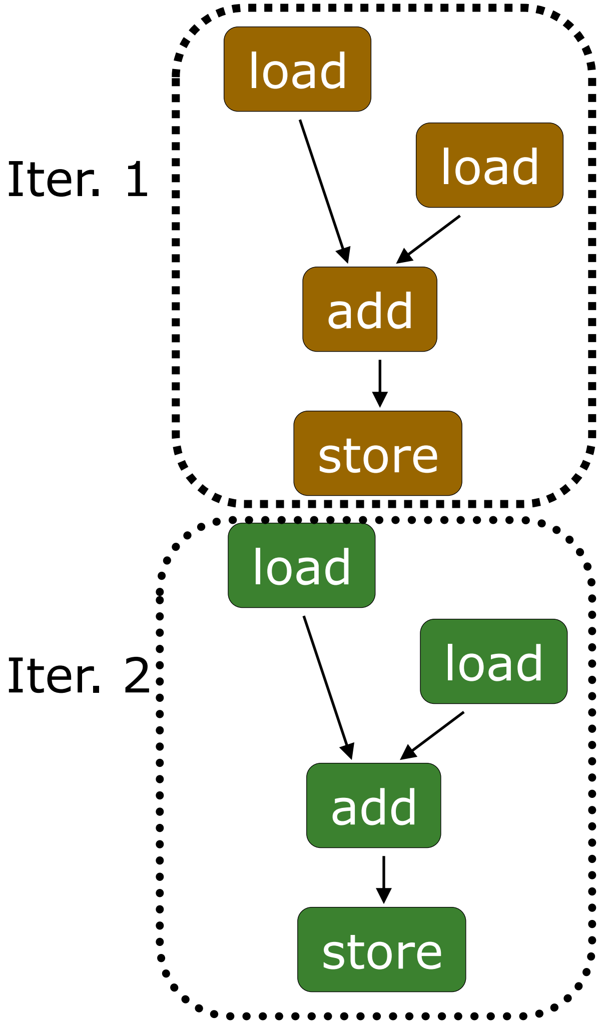
- Execution Model refers to how the hardware executes the code underneath
  - E.g., Out-of-order execution, Vector processor, Array processor, Dataflow processor, Multiprocessor, Multithreaded processor, …

- 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)

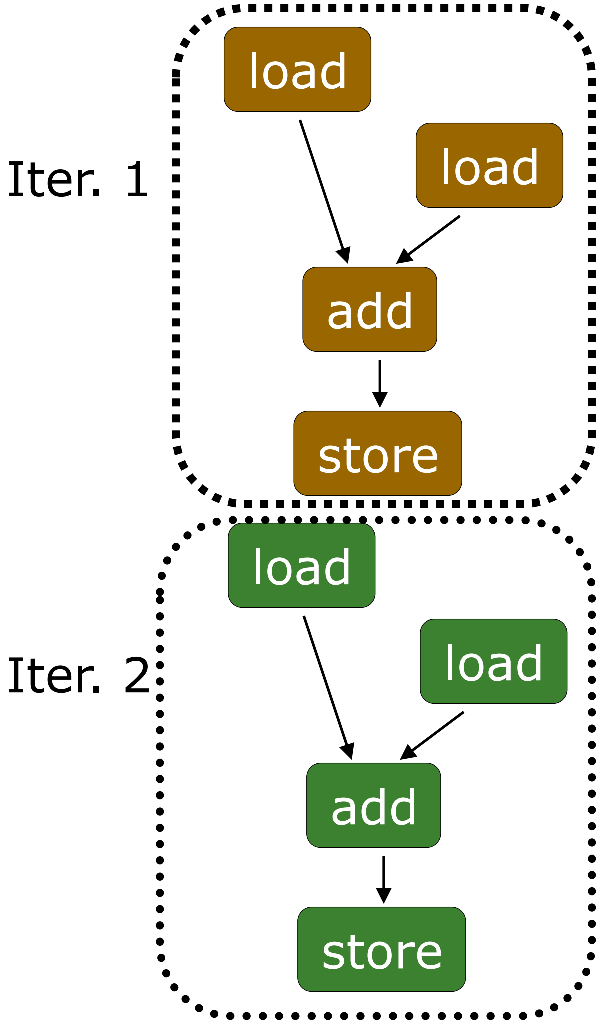
2. Data-Parallel (SIMD)

3. Multithreaded (MIMD/SPMD)

Prog. Model 1: Sequential (SISD)

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

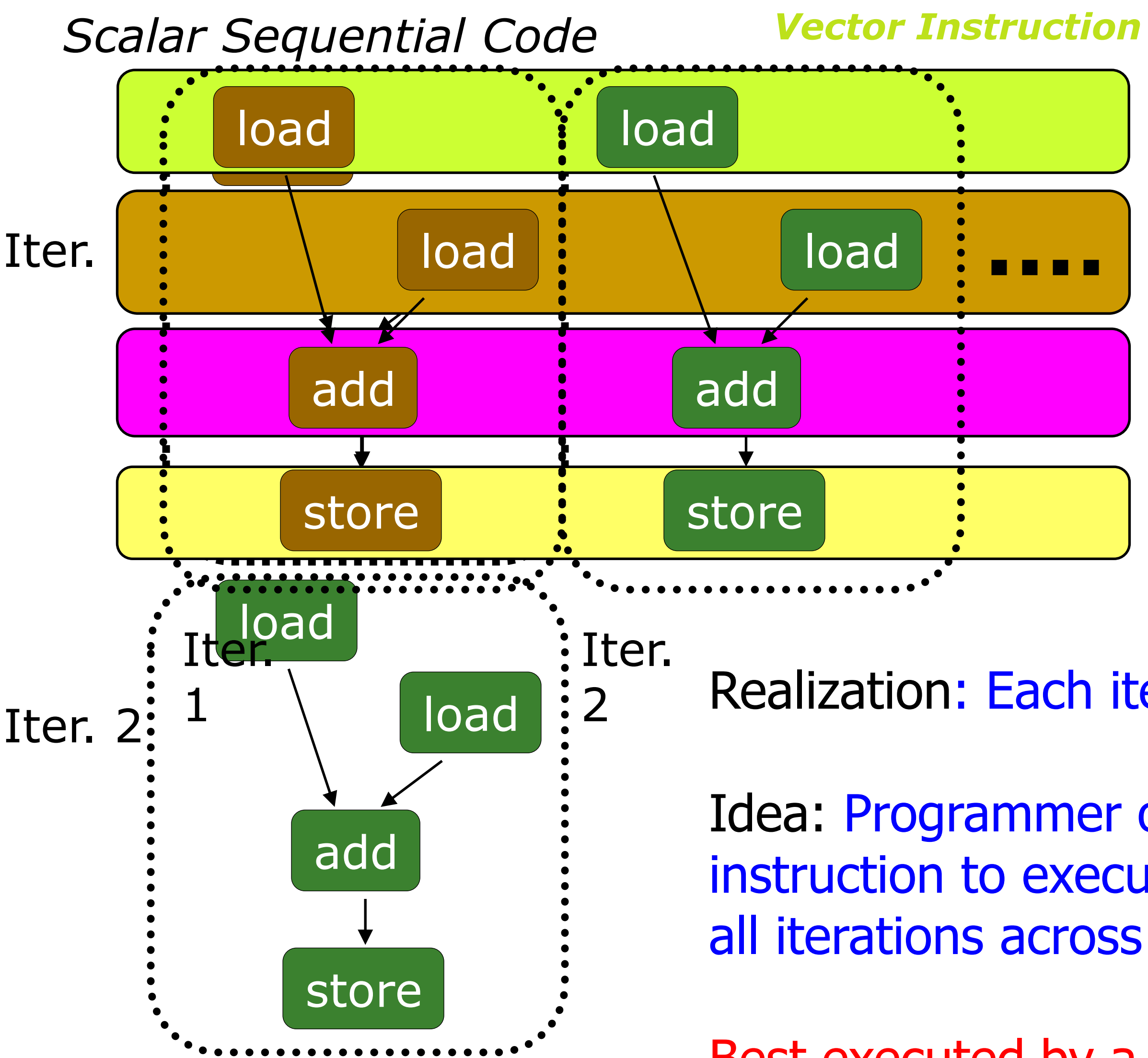
Scalar Sequential Code



- 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

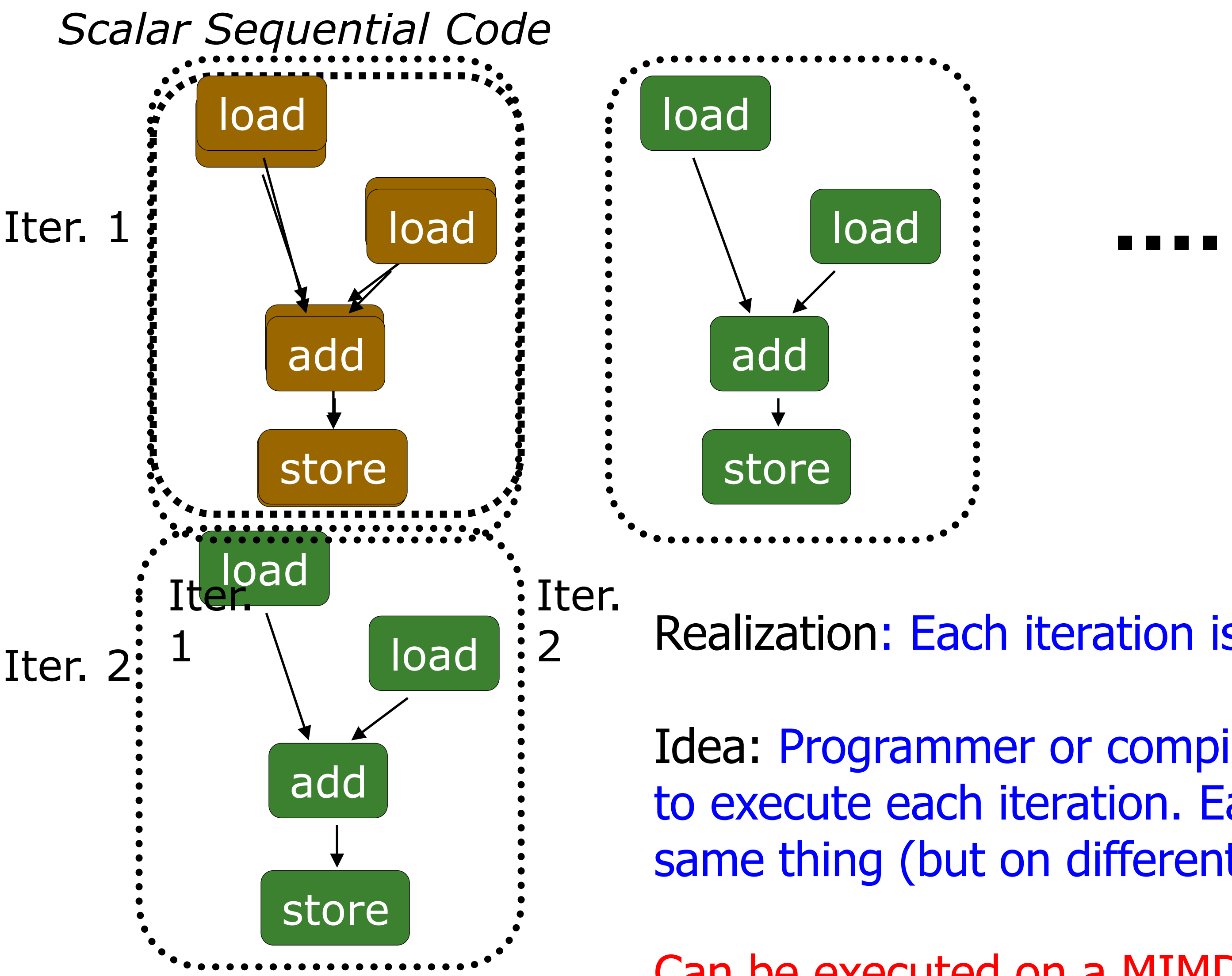
Realization: Each iteration is independent

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];
```



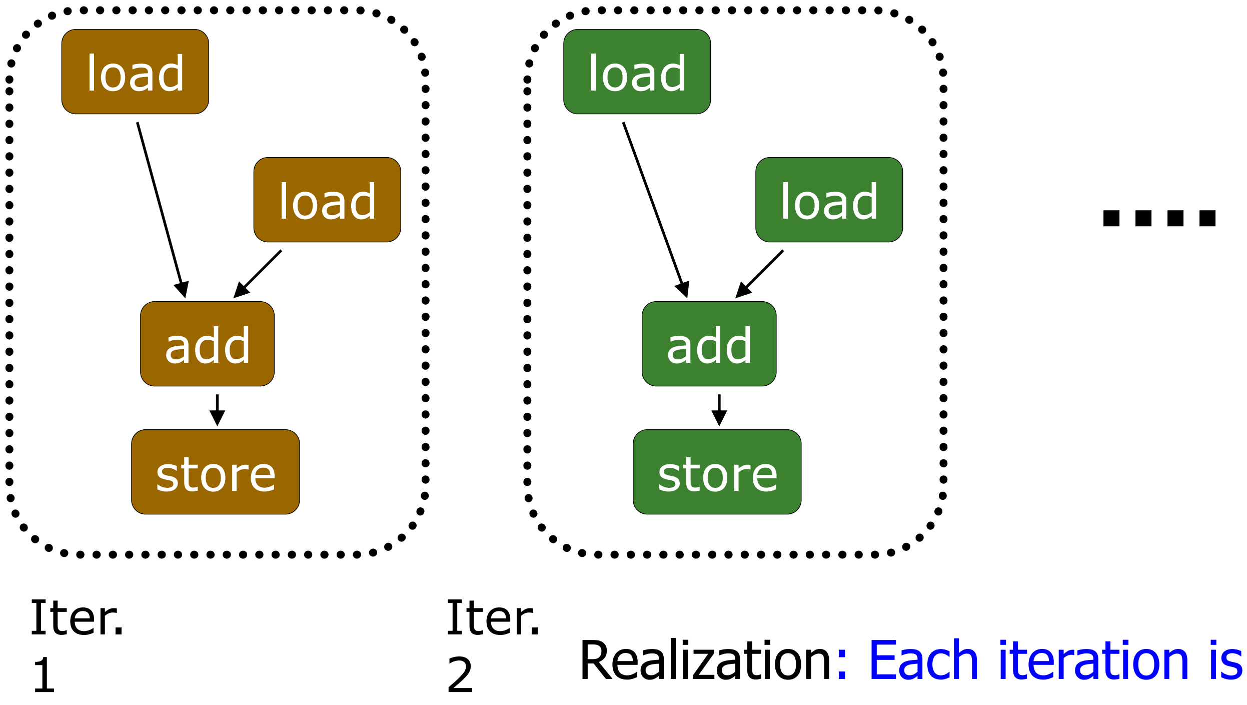
Realization: Each iteration is independent

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

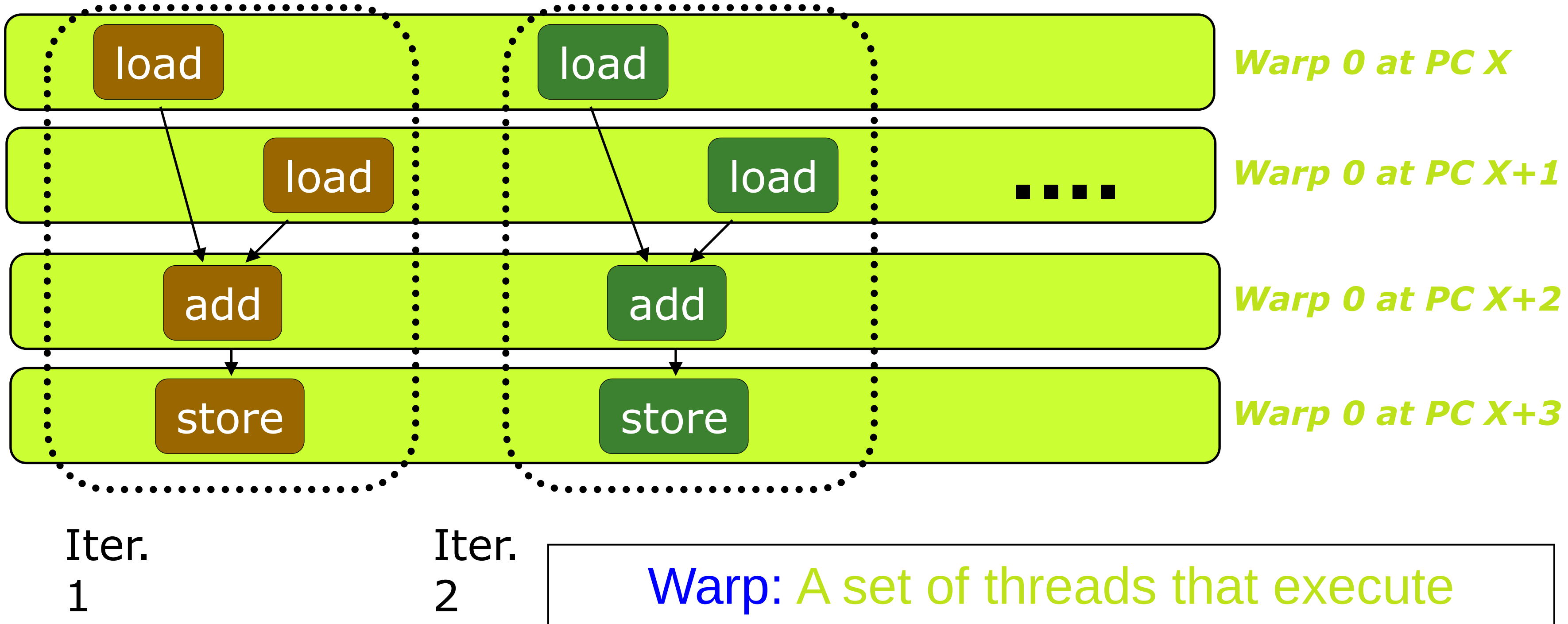
- Except it is **not** programmed using SIMD instructions

- 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

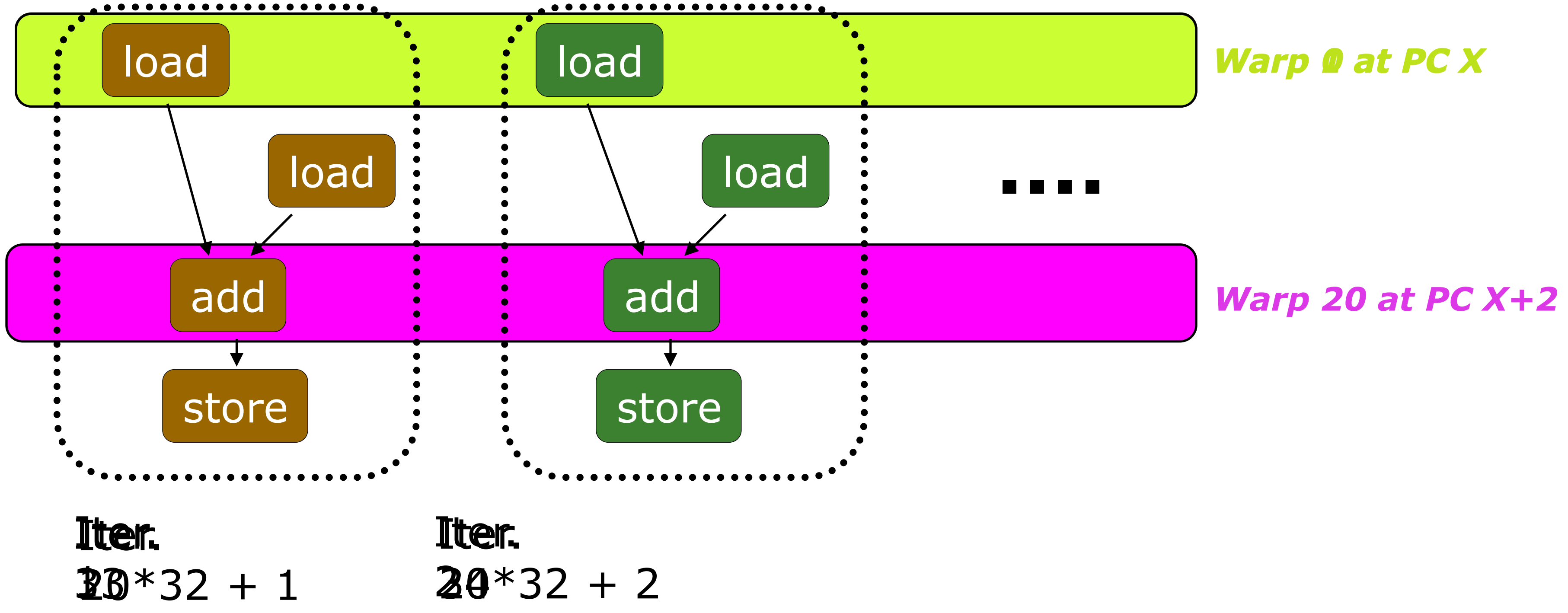
- SIMT: Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

- 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

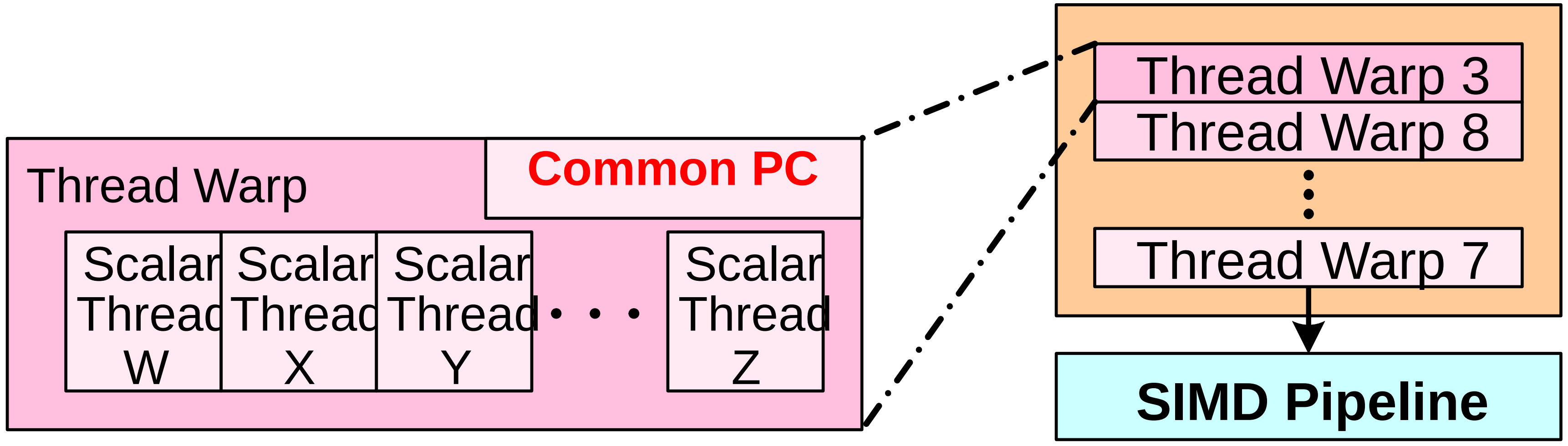
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

- 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



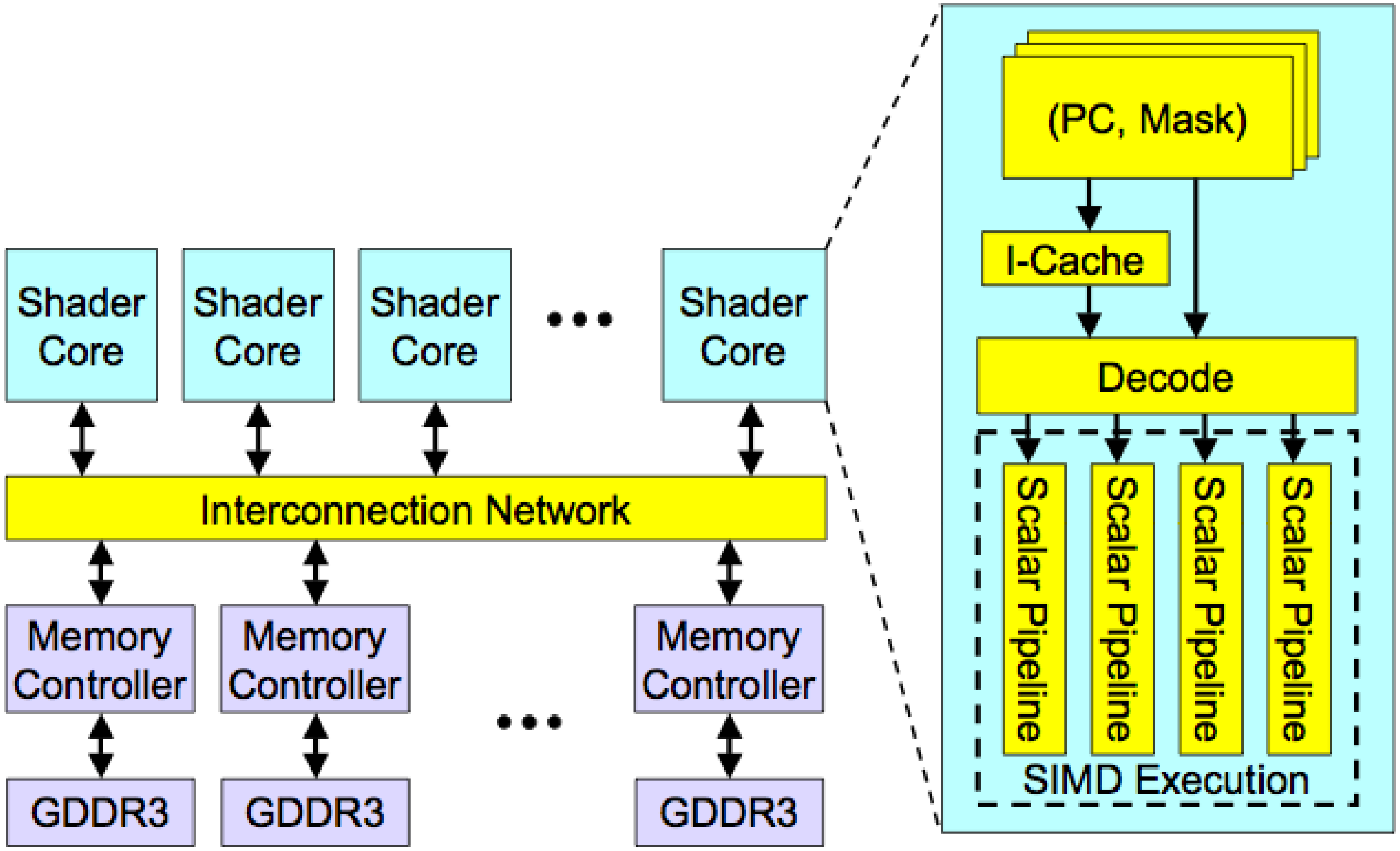
Warps and Warp-Level FGMT

- 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.

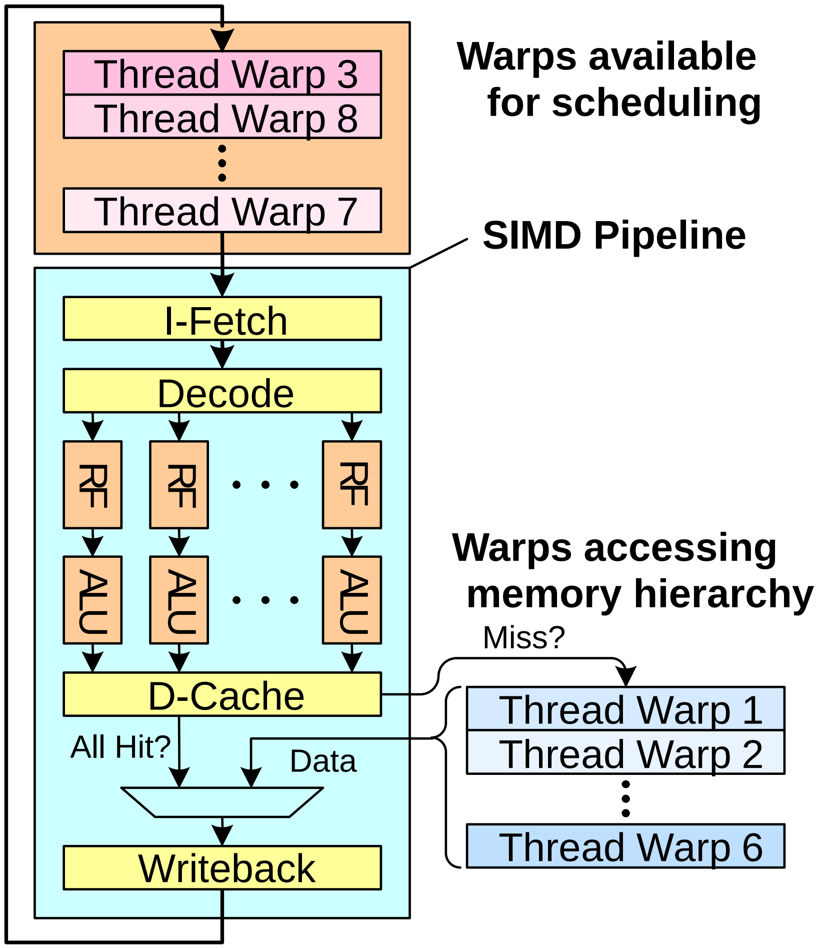
High-Level View of a GPU



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

Latency Hiding via Warp-Level FGMT

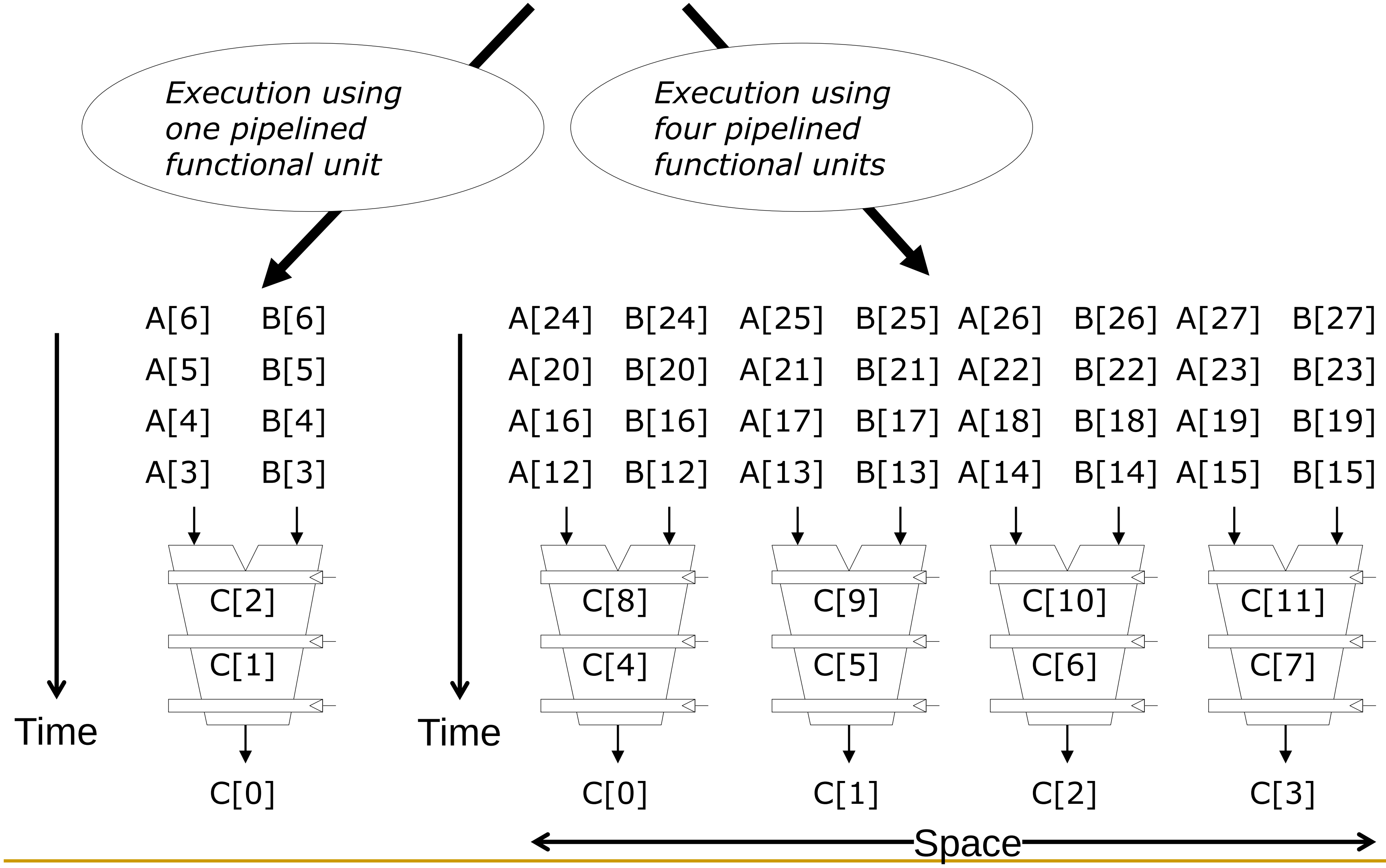
- Warp: A set of threads that execute the same instruction (on different data elements)
- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No interlocking)
  - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- FGMT enables long latency tolerance
  - Millions of pixels



Slide credit: Tor Aamodt

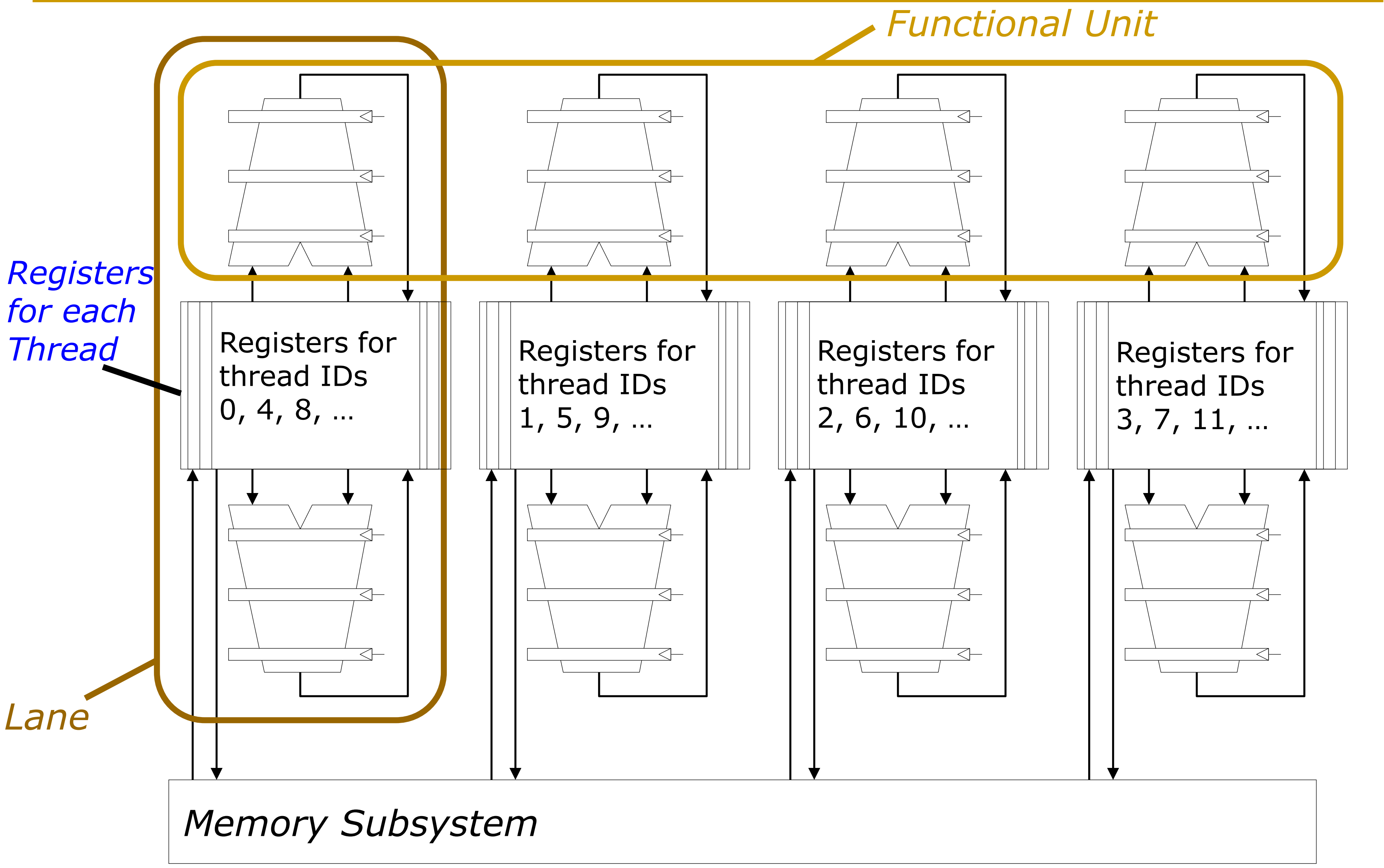
Warp Execution (Recall the Slide)

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


Execution using one pipelined functional unit
Execution using four pipelined functional units
A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]
C[2]
C[1]
C[0]
Time
A[24] B[24] A[25] B[25] A[26] B[26] A[27] B[27]
A[20] B[20] A[21] B[21] A[22] B[22] A[23] B[23]
A[16] B[16] A[17] B[17] A[18] B[18] A[19] B[19]
A[12] B[12] A[13] B[13] A[14] B[14] A[15] B[15]
C[8] C[9] C[10] C[11]
C[4] C[5] C[6] C[7]
C[0] C[1] C[2] C[3]
Time
Space

Slide credit: Krste Asanovic

SIMD Execution Unit Structure

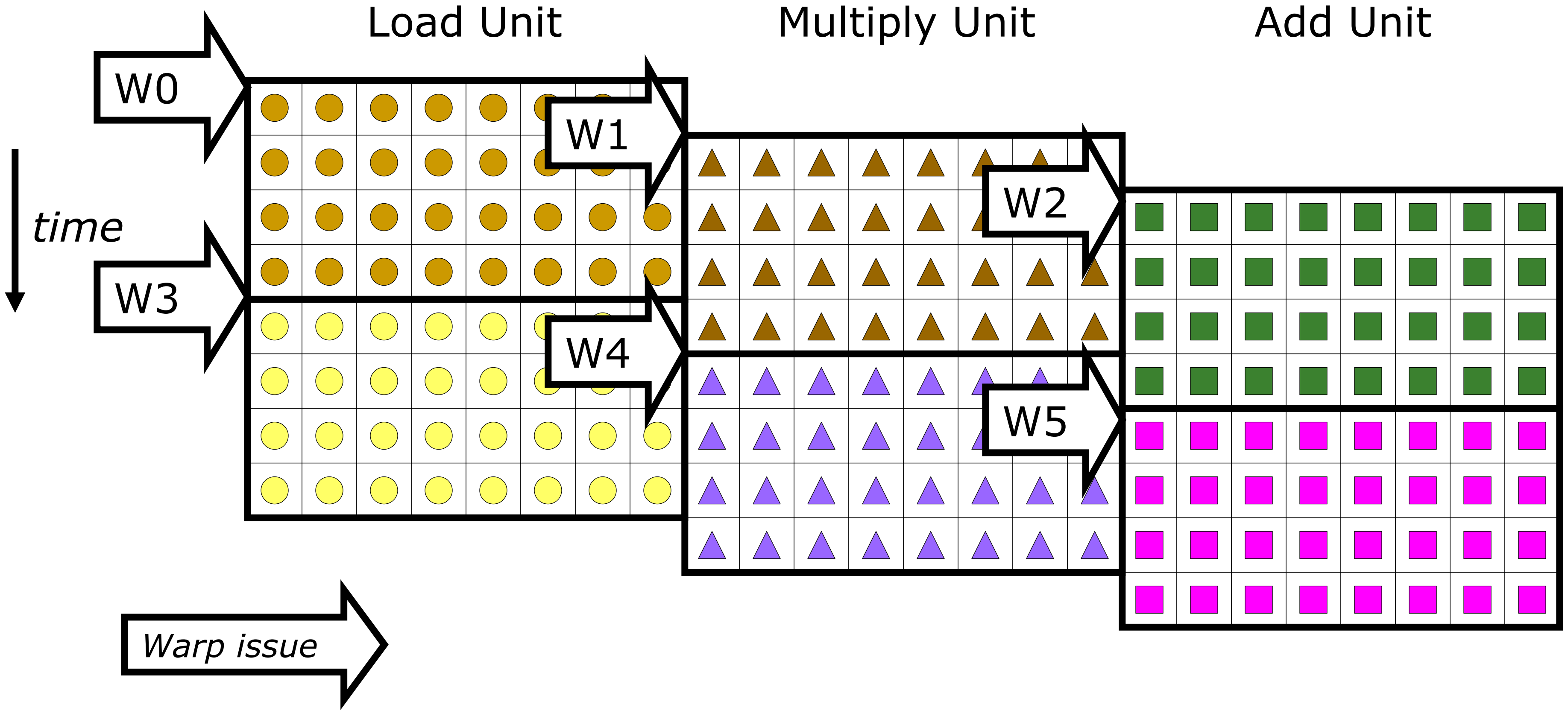

Functional Unit
Registers for each Thread
Registers for thread IDs 0, 4, 8, …
Registers for thread IDs 1, 5, 9, …
Registers for thread IDs 2, 6, 10, …
Registers for thread IDs 3, 7, 11, …
Lane
Memory Subsystem

Slide credit: Krste Asanovic

Warp Instruction Level Parallelism

Can overlap execution of multiple instructions

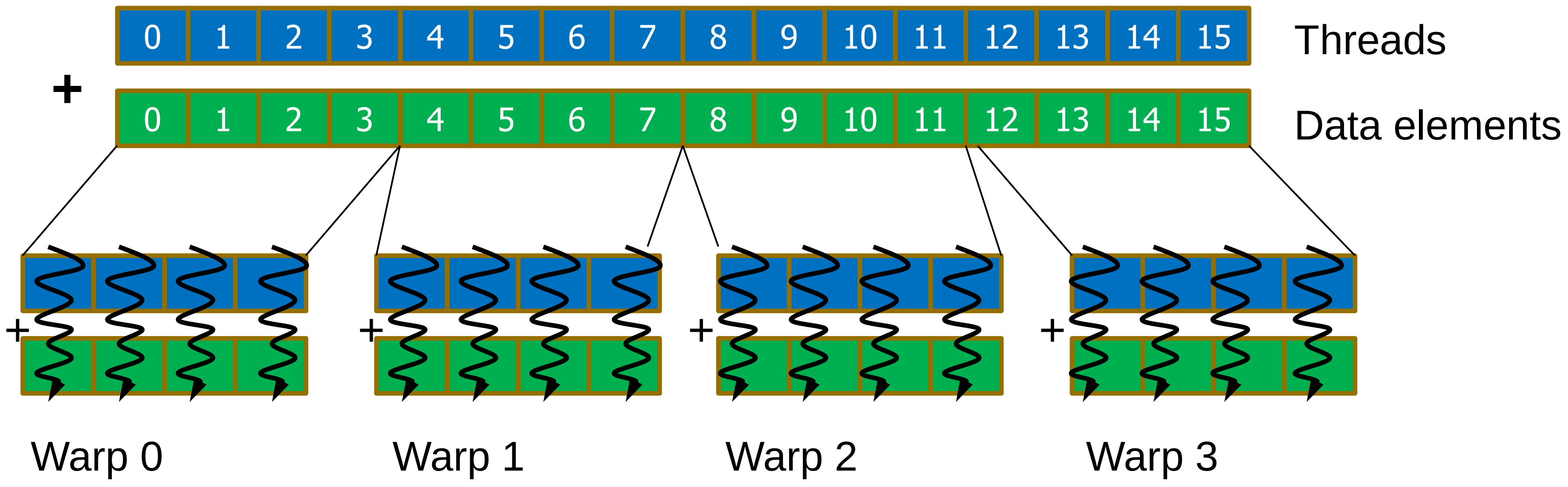
- Example machine has 32 threads per warp and 8 lanes
- Completes 24 operations/cycle while issuing 1 warp/cycle



SIMT Memory Access

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

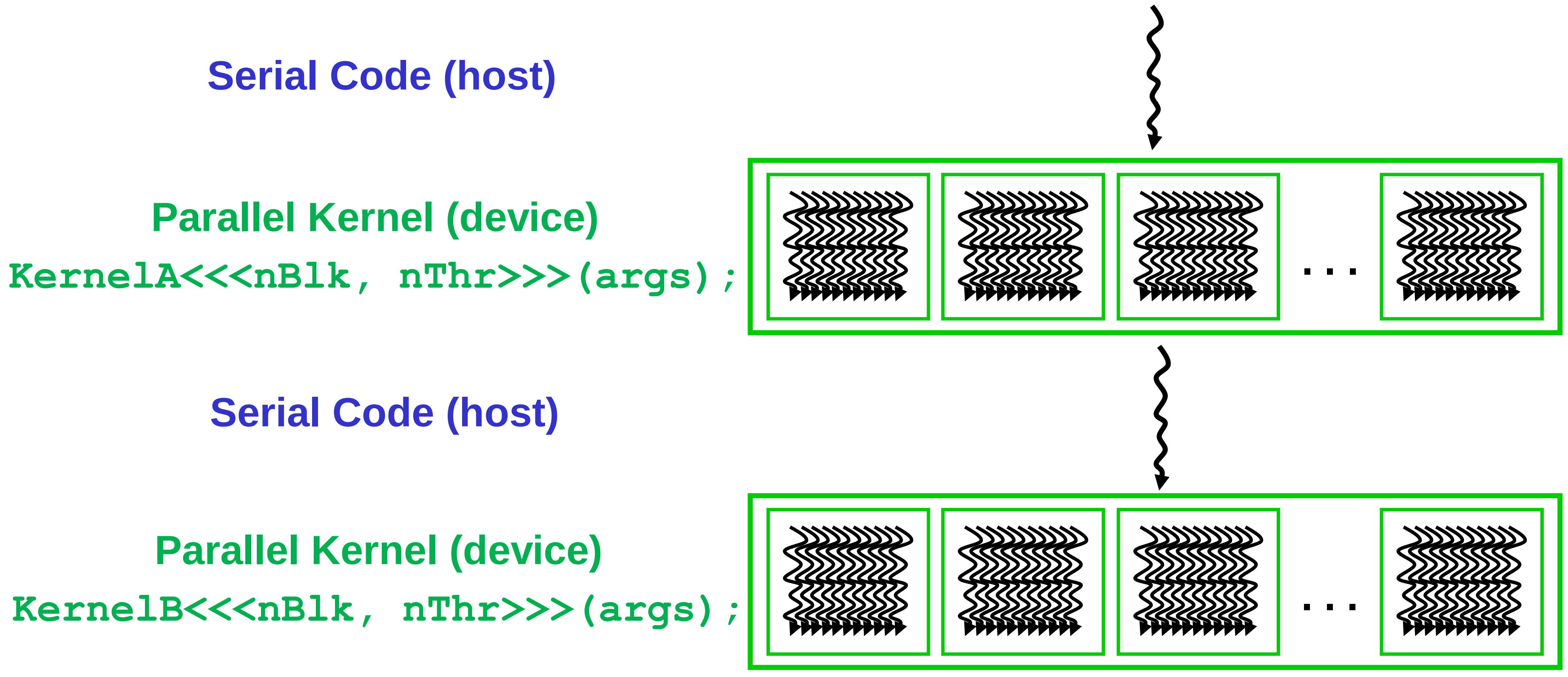
Let's assume N=16, 4 threads per warp → 4 warps



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



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;
}
```

Slide credit: Hyesoon Kim

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);
}
```

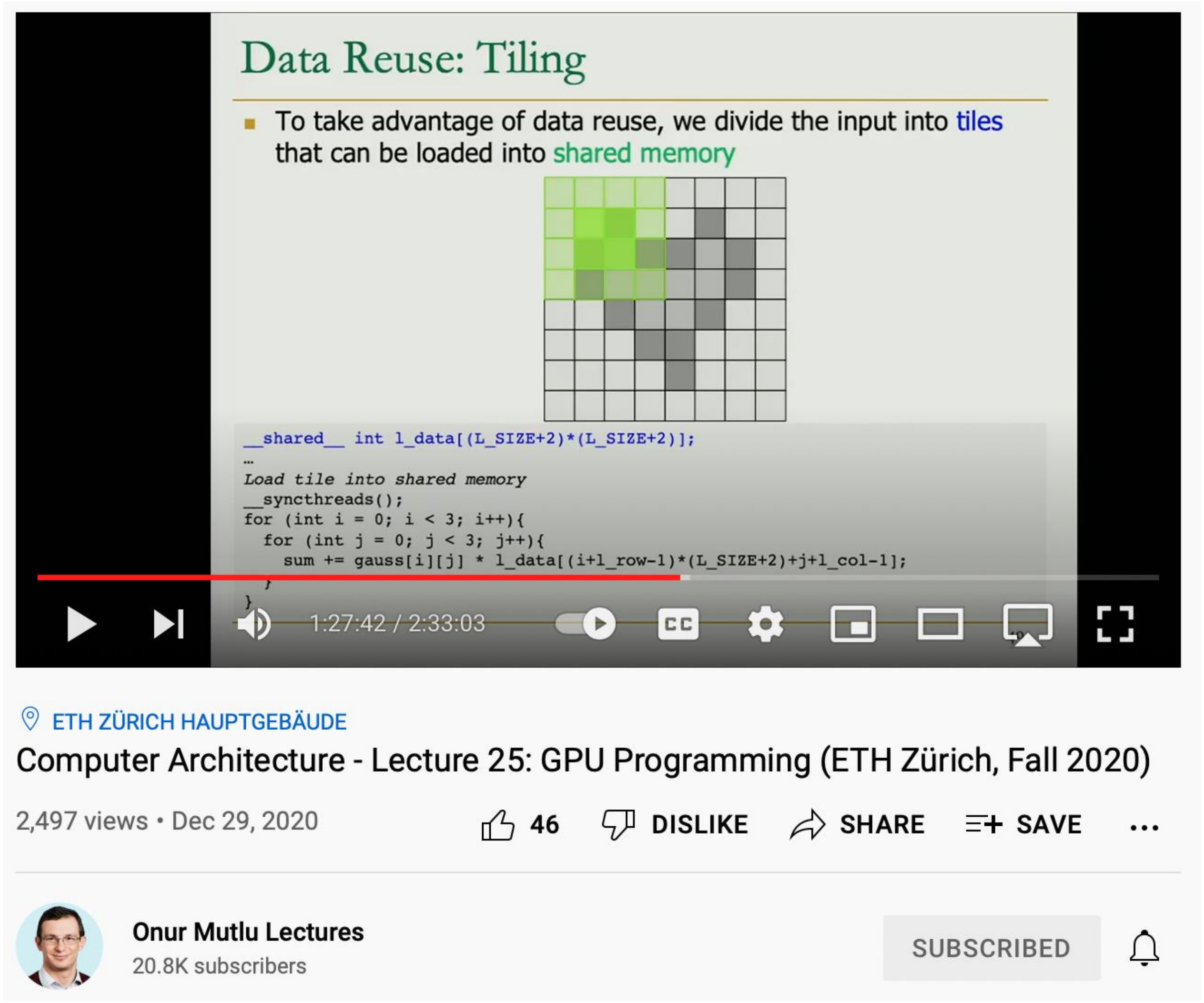
GPU Program

```
__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



Data Reuse: Tiling

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

```
__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];
  }
}
```

1:27:42 / 2:33:03

ETH ZÜRICH HAUPTGEBÄUDE

Computer Architecture - Lecture 25: GPU Programming (ETH Zürich, Fall 2020)

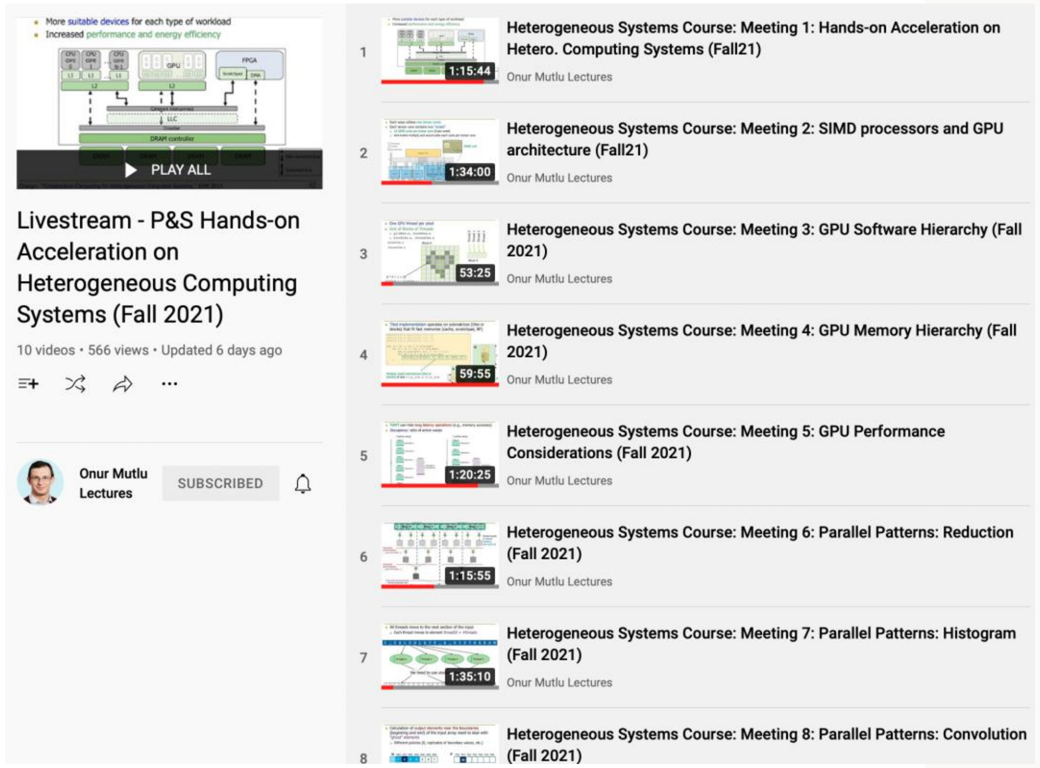
2,497 views • Dec 29, 2020

Onur Mutlu Lectures
20.8K subscribers

https://youtu.be/AkYnuqVpCug

Heterogeneous Systems Course (Fall 2021)

- Short weekly lectures
- Hands-on projects

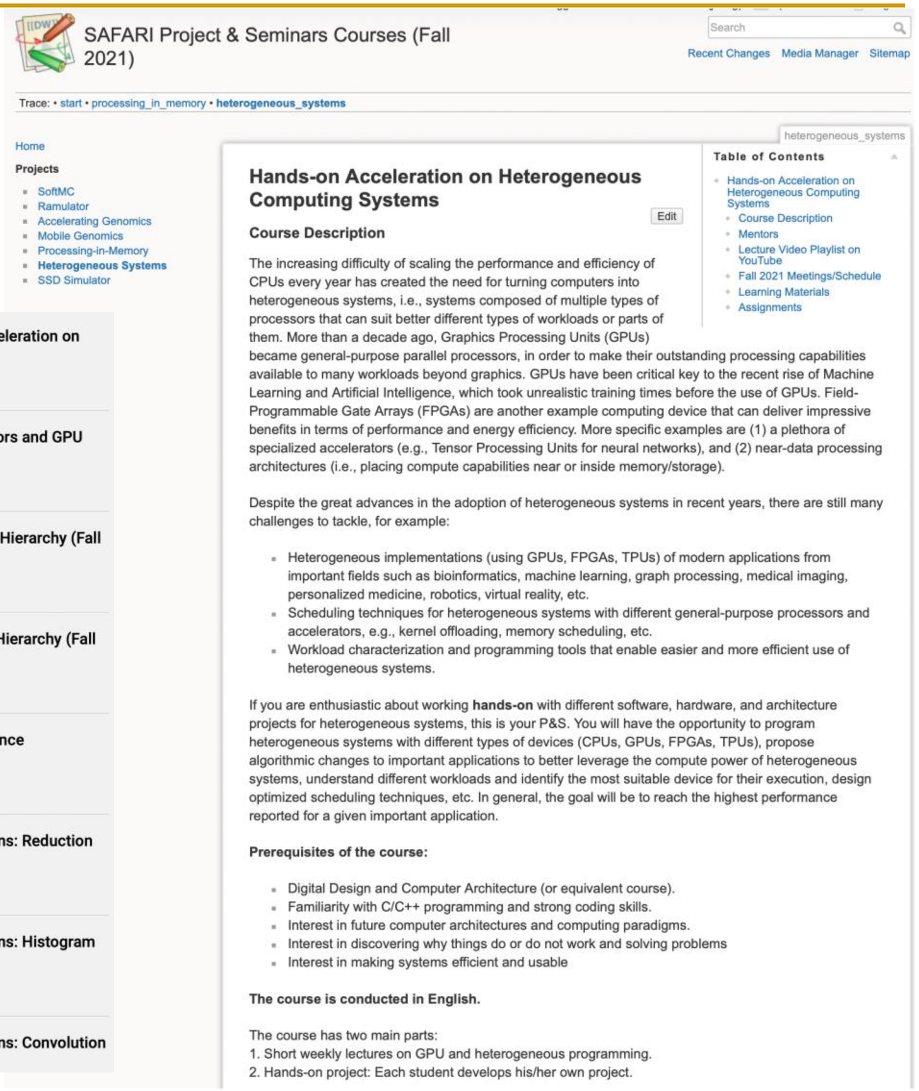

Livestream - P&S Hands-on Acceleration on Heterogeneous Computing Systems (Fall 2021)

10 videos • 566 views • Updated 6 days ago

Onur Mutlu Lectures SUBSCRIBED

1. Heterogeneous Systems Course: Meeting 1: Hands-on Acceleration on Hetero. Computing Systems (Fall21) — 1:15:44 — Onur Mutlu Lectures
2. Heterogeneous Systems Course: Meeting 2: SIMD processors and GPU architecture (Fall21) — 1:34:00 — Onur Mutlu Lectures
3. Heterogeneous Systems Course: Meeting 3: GPU Software Hierarchy (Fall 2021) — 53:25 — Onur Mutlu Lectures
4. Heterogeneous Systems Course: Meeting 4: GPU Memory Hierarchy (Fall 2021) — 59:55 — Onur Mutlu Lectures
5. Heterogeneous Systems Course: Meeting 5: GPU Performance Considerations (Fall 2021) — 1:20:25 — Onur Mutlu Lectures
6. Heterogeneous Systems Course: Meeting 6: Parallel Patterns: Reduction (Fall 2021) — 1:15:55 — Onur Mutlu Lectures
7. Heterogeneous Systems Course: Meeting 7: Parallel Patterns: Histogram (Fall 2021) — 1:35:10 — Onur Mutlu Lectures
8. Heterogeneous Systems Course: Meeting 8: Parallel Patterns: Convolution (Fall 2021)

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


SAFARI Project & Seminars Courses (Fall 2021)

Search | Recent Changes | Media Manager | Sitemap

Trace: • start • processing_in_memory • heterogeneous_systems

Home

Projects
- SoftMC
- Ramulator
- Accelerating Genomics
- Mobile Genomics
- Processing-in-Memory
- **Heterogeneous Systems**
- SSD Simulator

Table of Contents
- Hands-on Acceleration on Heterogeneous Computing Systems
  - Course Description
  - Mentors
  - Lecture Video Playlist on YouTube
  - Fall 2021 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.

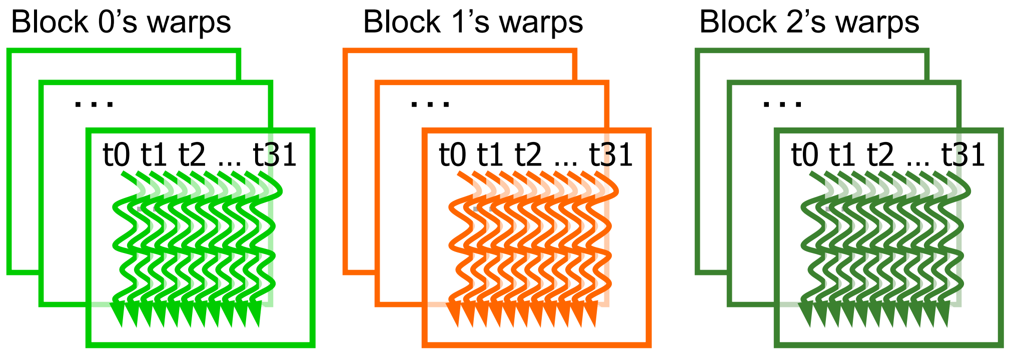
https://youtube.com/playlist?list=PL5Q2soXY2Zi_OwkTgEyA6tk3UsoPBH737

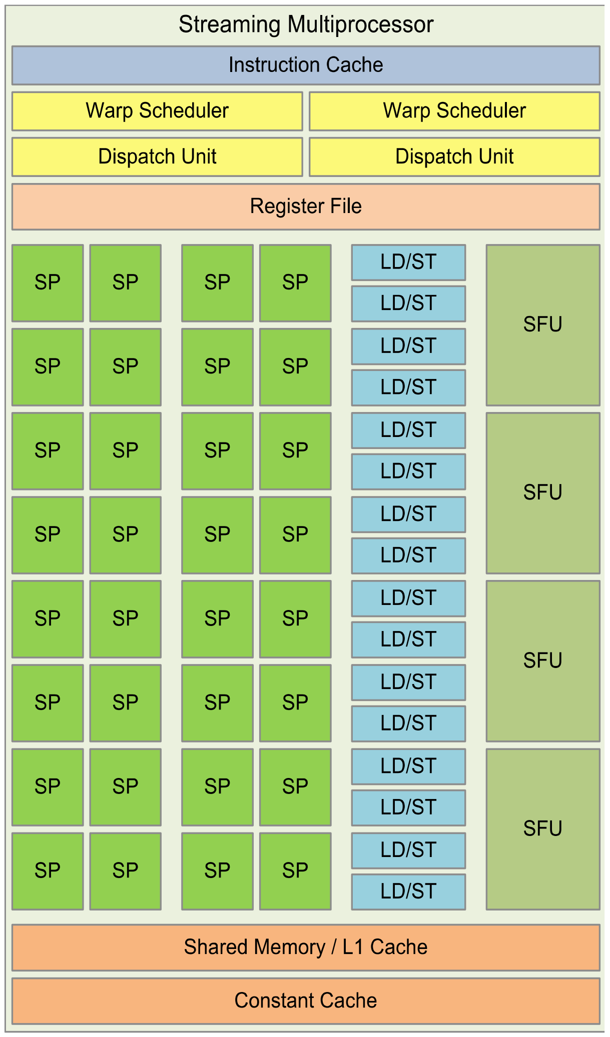
https://safari.ethz.ch/projects_and_seminars/fall2021/doku.php?id=heterogeneous_systems

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

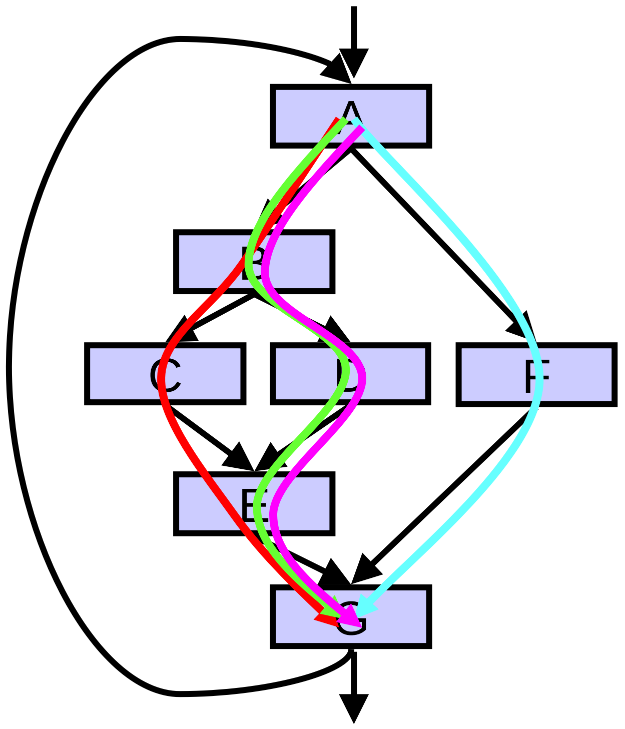
- Essentially, multiple instruction streams execute the same program
  - Each program/procedure 1) works on different data, 2) can execute a different control-flow path, at run-time
  - Many scientific applications are programmed this way and run on MIMD hardware (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD hardware

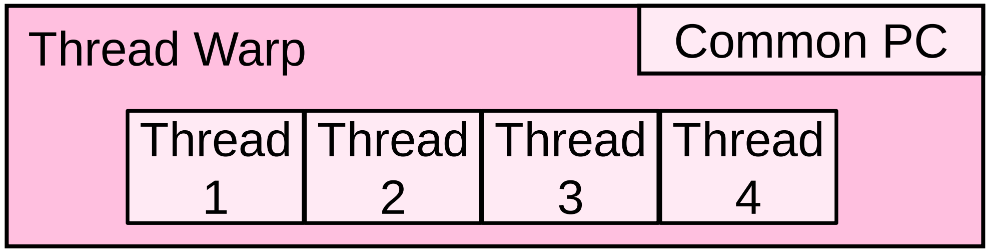
SIMD vs. SIMT Execution Model

- 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

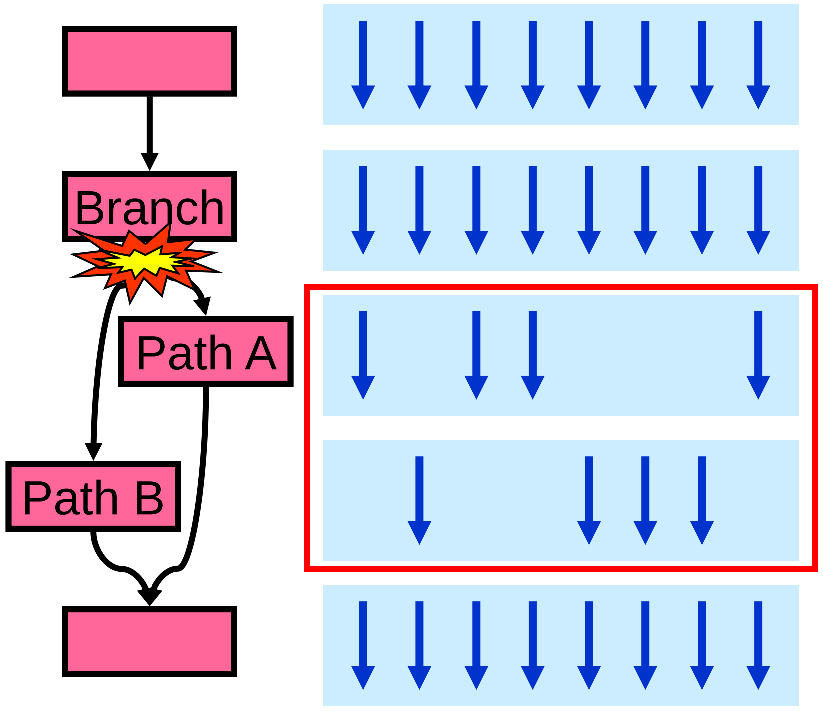




Control Flow Problem in GPUs/SIMT

- 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?

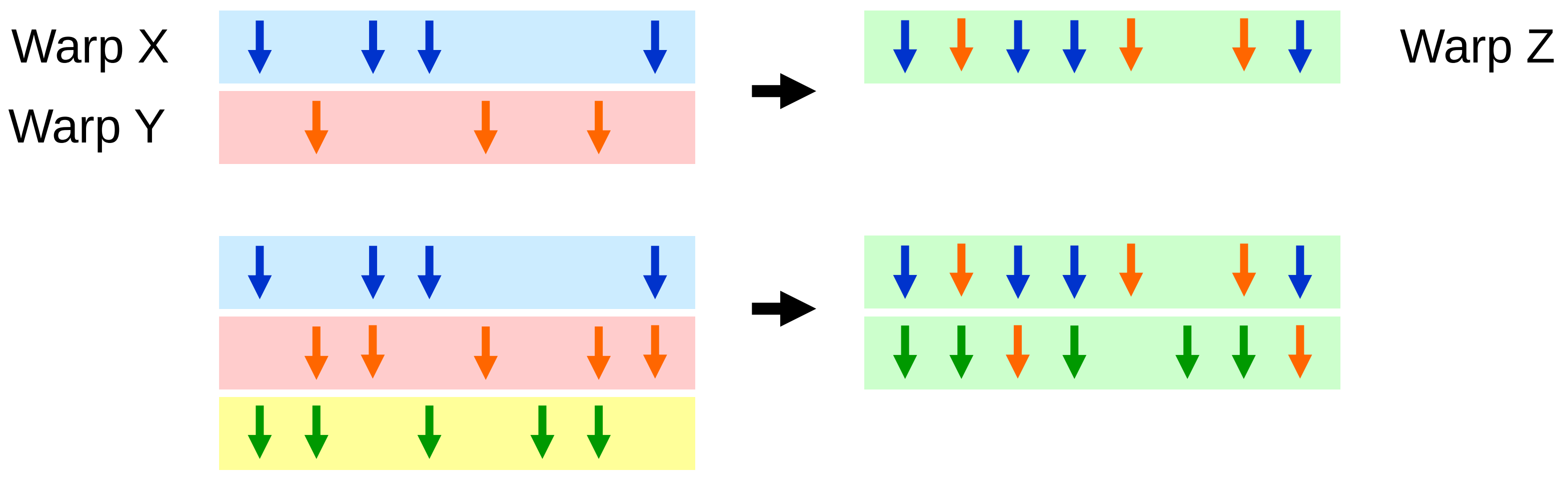
Remember: Each Thread Is Independent

- 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)

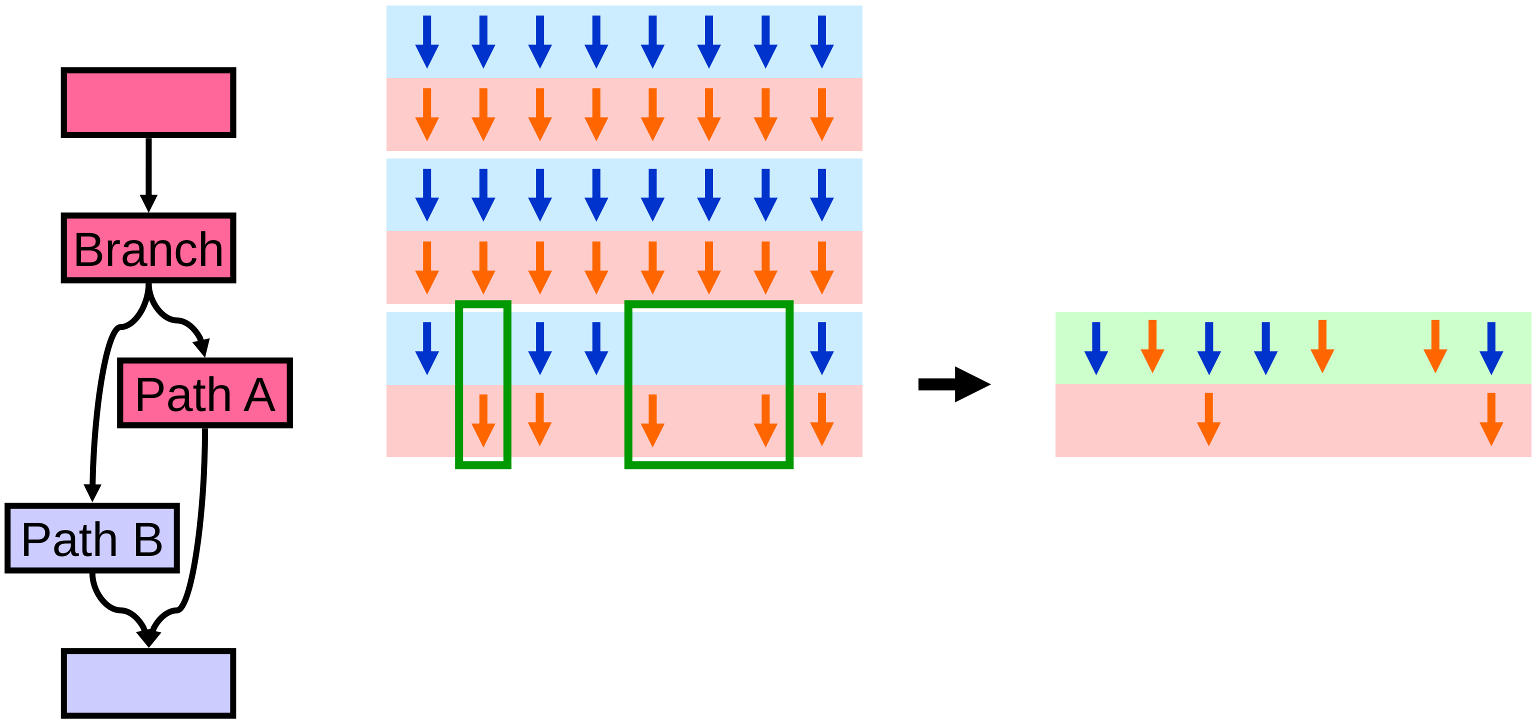
Dynamic Warp Formation/Merging

- 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



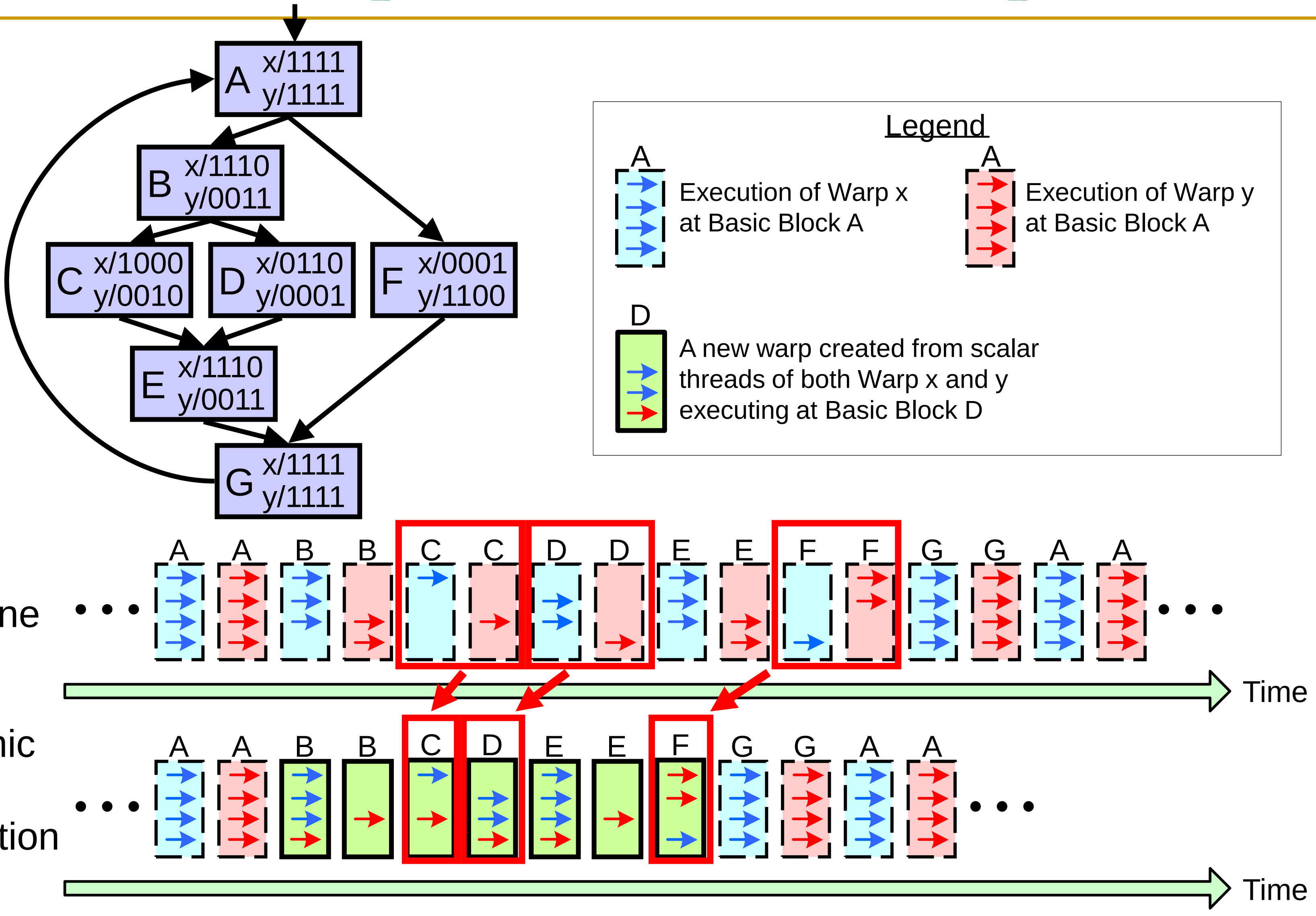
Dynamic Warp Formation/Merging

- 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.

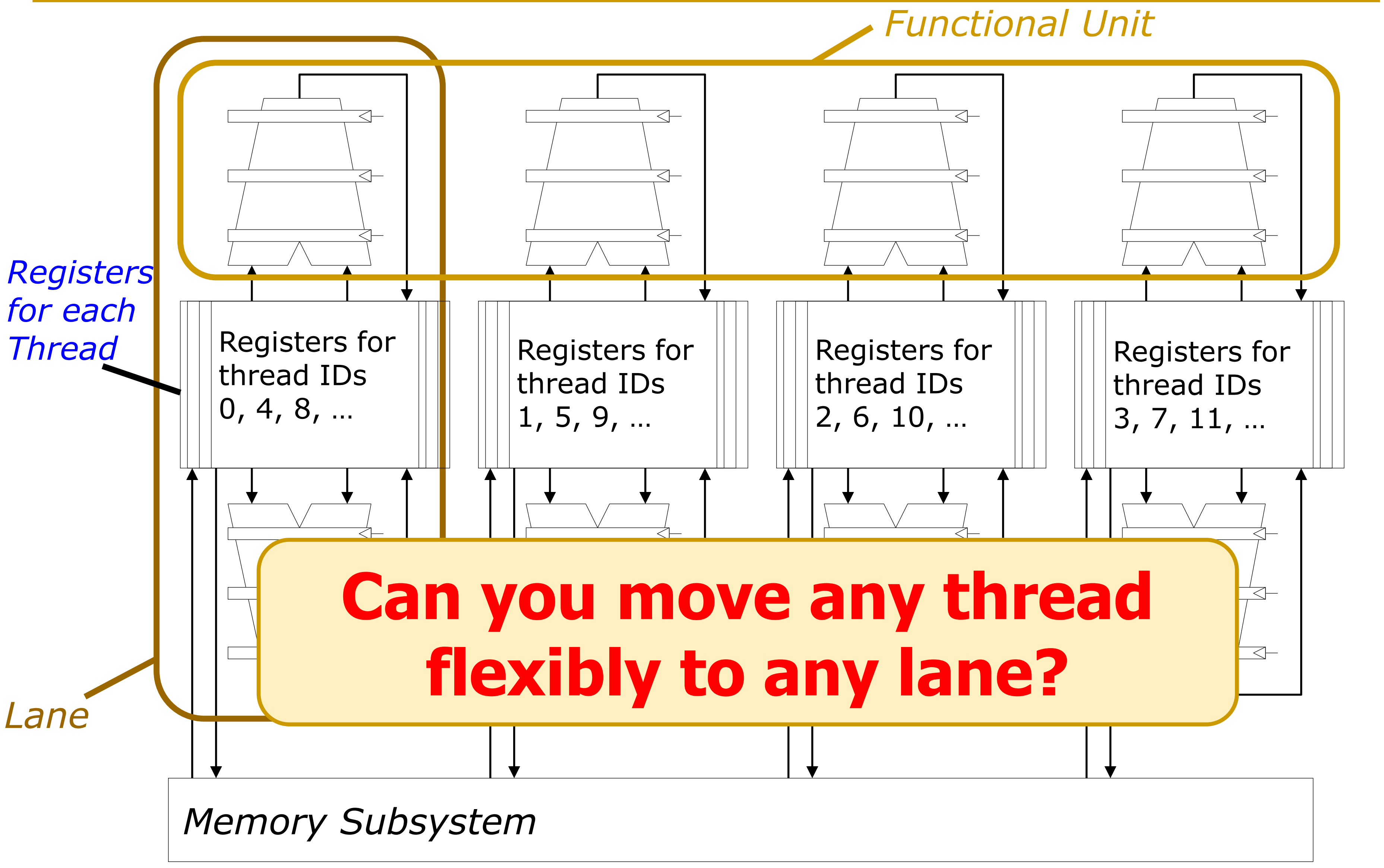
Dynamic Warp Formation Example



A x/1111 y/1111
B x/1110 y/0011
C x/1000 y/0010
D x/0110 y/0001
F x/0001 y/1100
E x/1110 y/0011
G x/1111 y/1111
Legend
A
Execution of Warp x at Basic Block A
A
Execution of Warp y at Basic Block A
D
A new warp created from scalar threads of both Warp x and y executing at Basic Block D
Baseline
A A B B C C D D E E F F G G A A
Time
Dynamic Warp Formation
A A B B C D E E F G G A A
Time

Slide credit: Tor Aamodt

Hardware Constraints Limit Flexibility of Warp Grouping

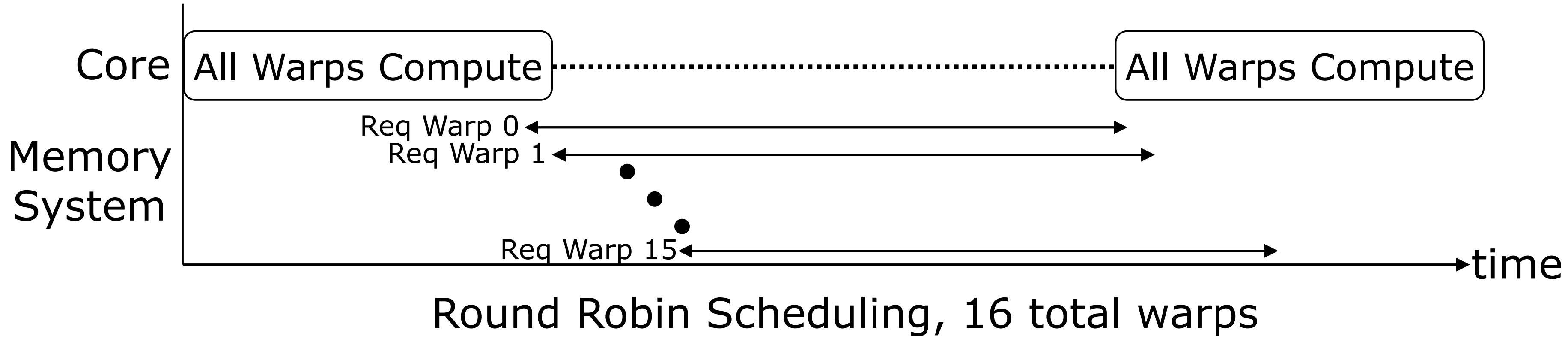

Functional Unit
Registers for each Thread
Registers for thread IDs 0, 4, 8, …
Registers for thread IDs 1, 5, 9, …
Registers for thread IDs 2, 6, 10, …
Registers for thread IDs 3, 7, 11, …
Lane
Memory Subsystem

Can you move any thread flexibly to any lane?

Slide credit: Krste Asanovic

Large Warps and Two-Level Warp Scheduling

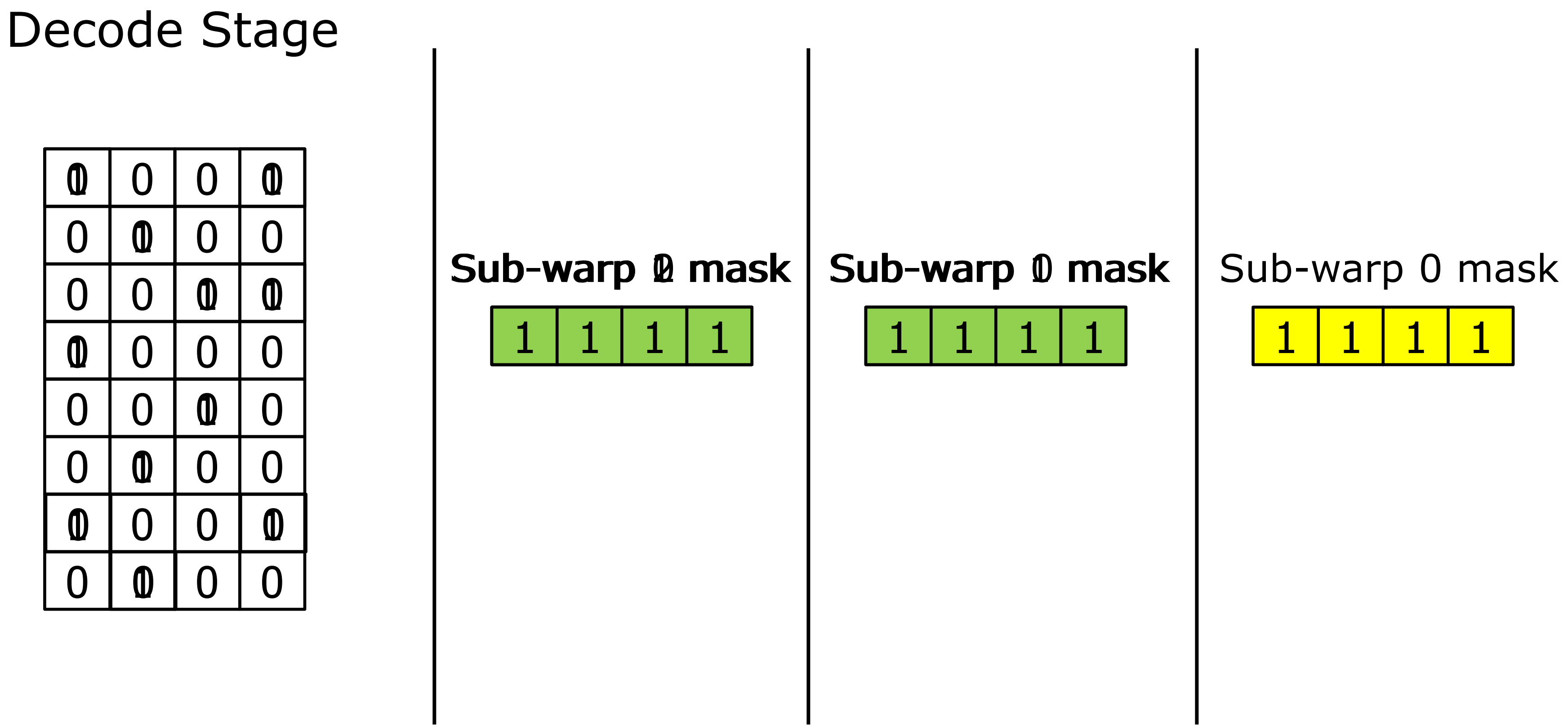
- Two main reasons for GPU resources be underutilized
  - Branch divergence
  - Long latency operations



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

Large Warp Microarchitecture Example

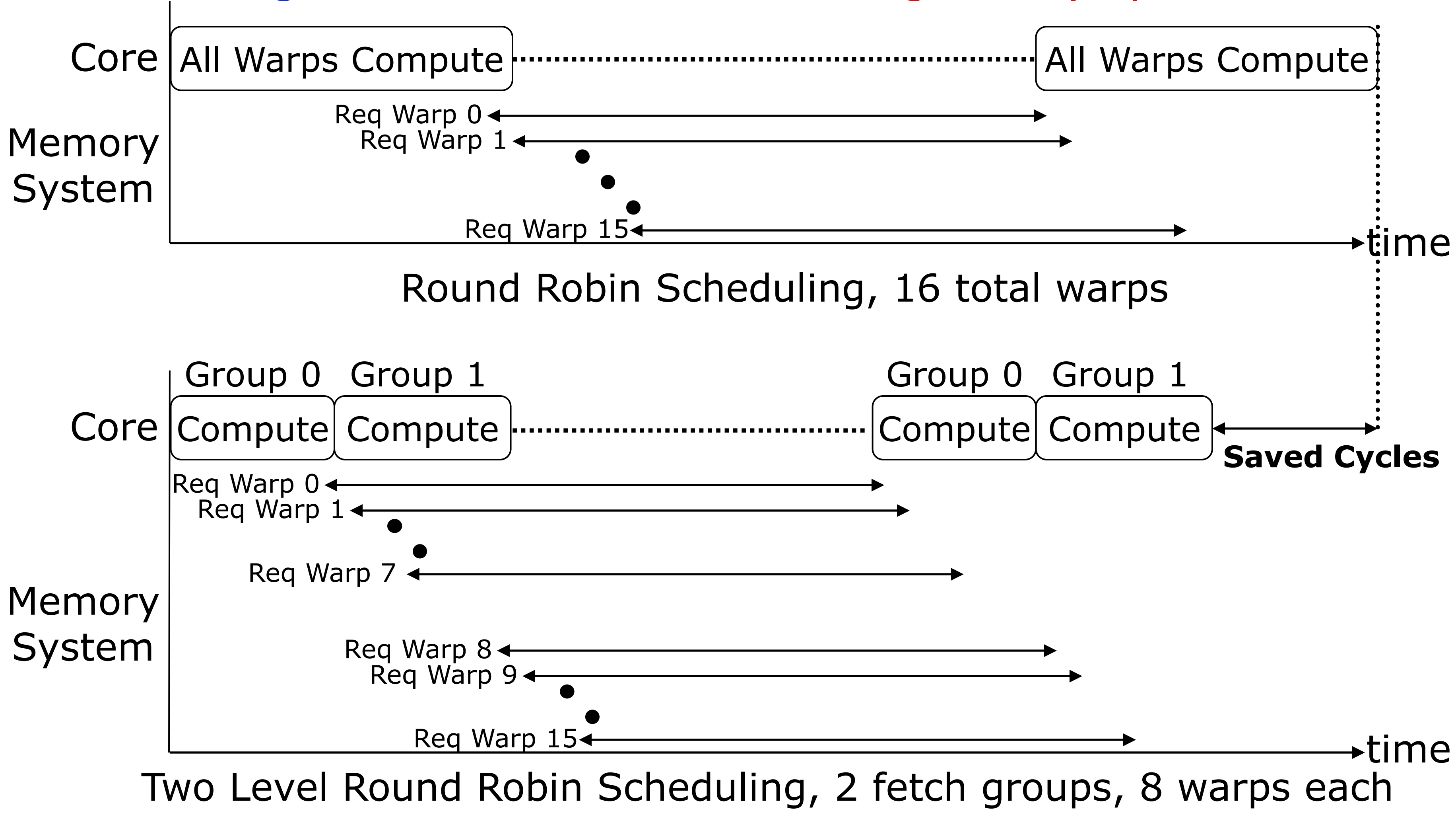
- Reduce branch divergence by having large warps
- Dynamically break down a large warp into sub-warps



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.

An Example GPU

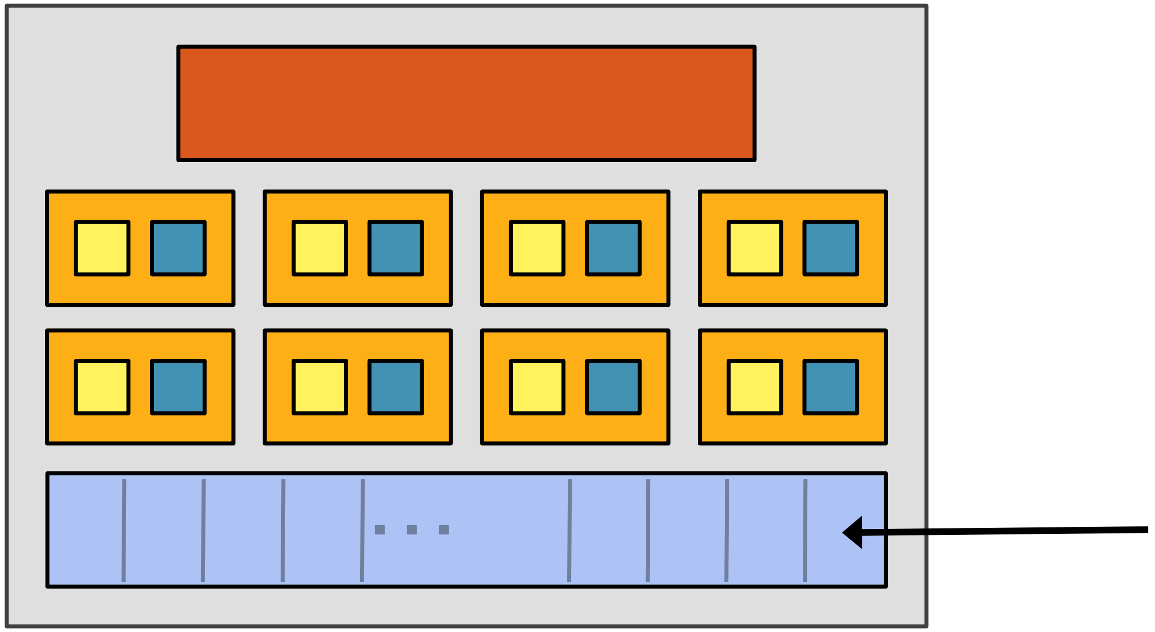
NVIDIA GeForce GTX 285

- 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

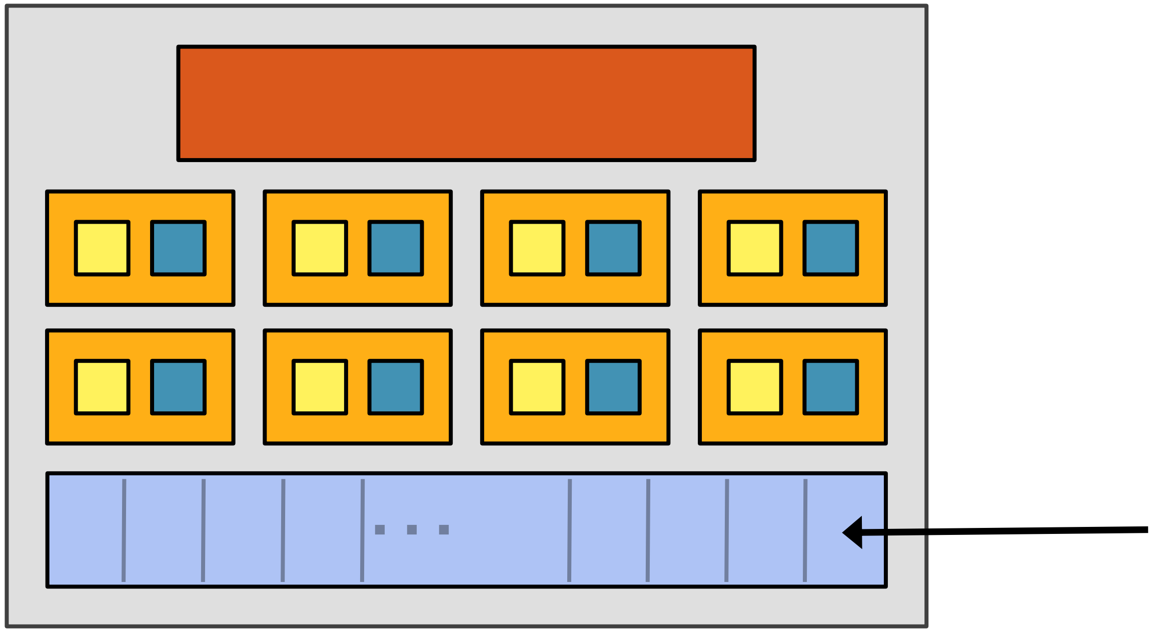
🟨 = multiply-add

🟦 = multiply

🟥 = instruction stream decode

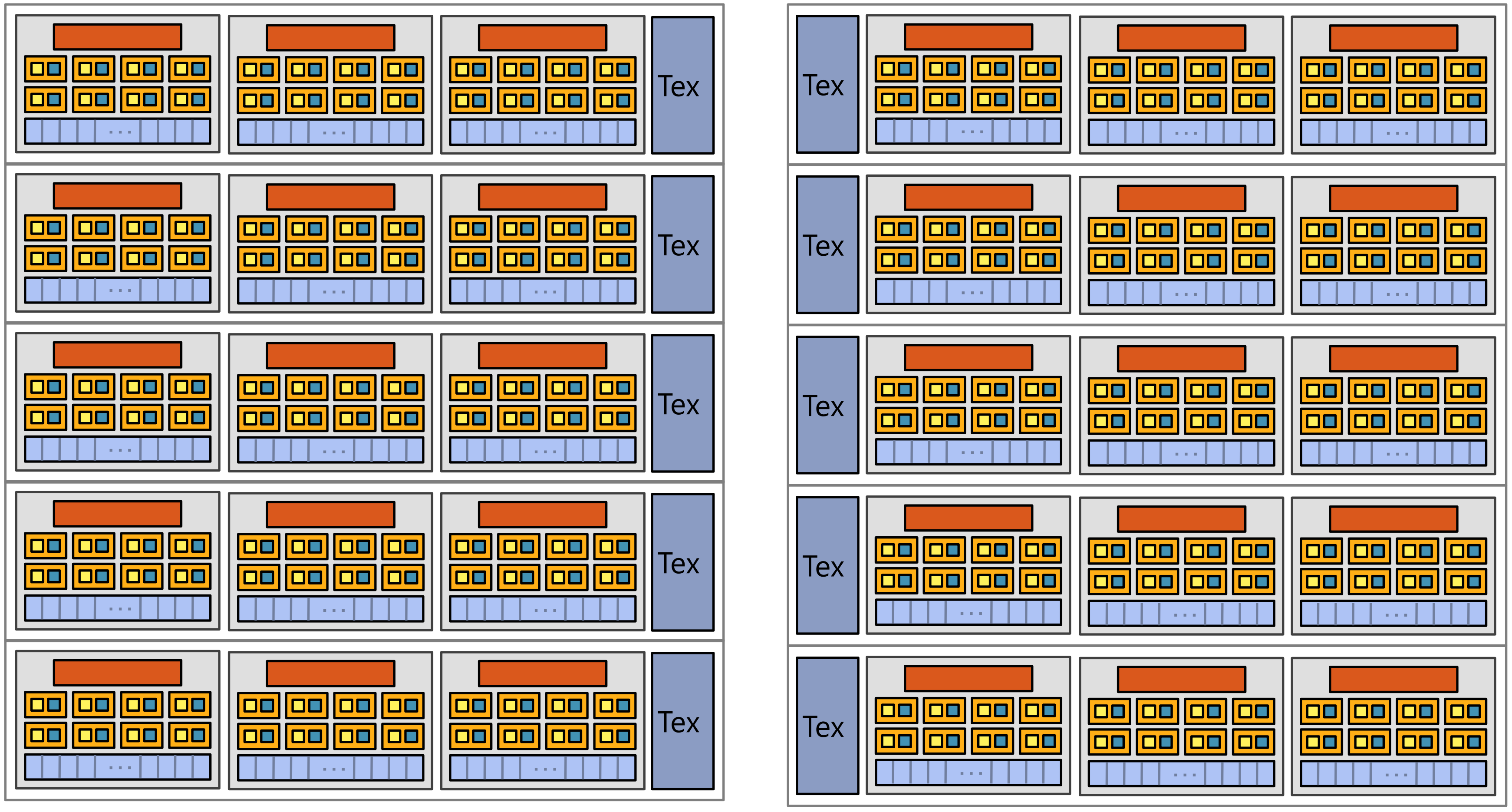
🟦 = execution context storage

NVIDIA GeForce GTX 285 "core"



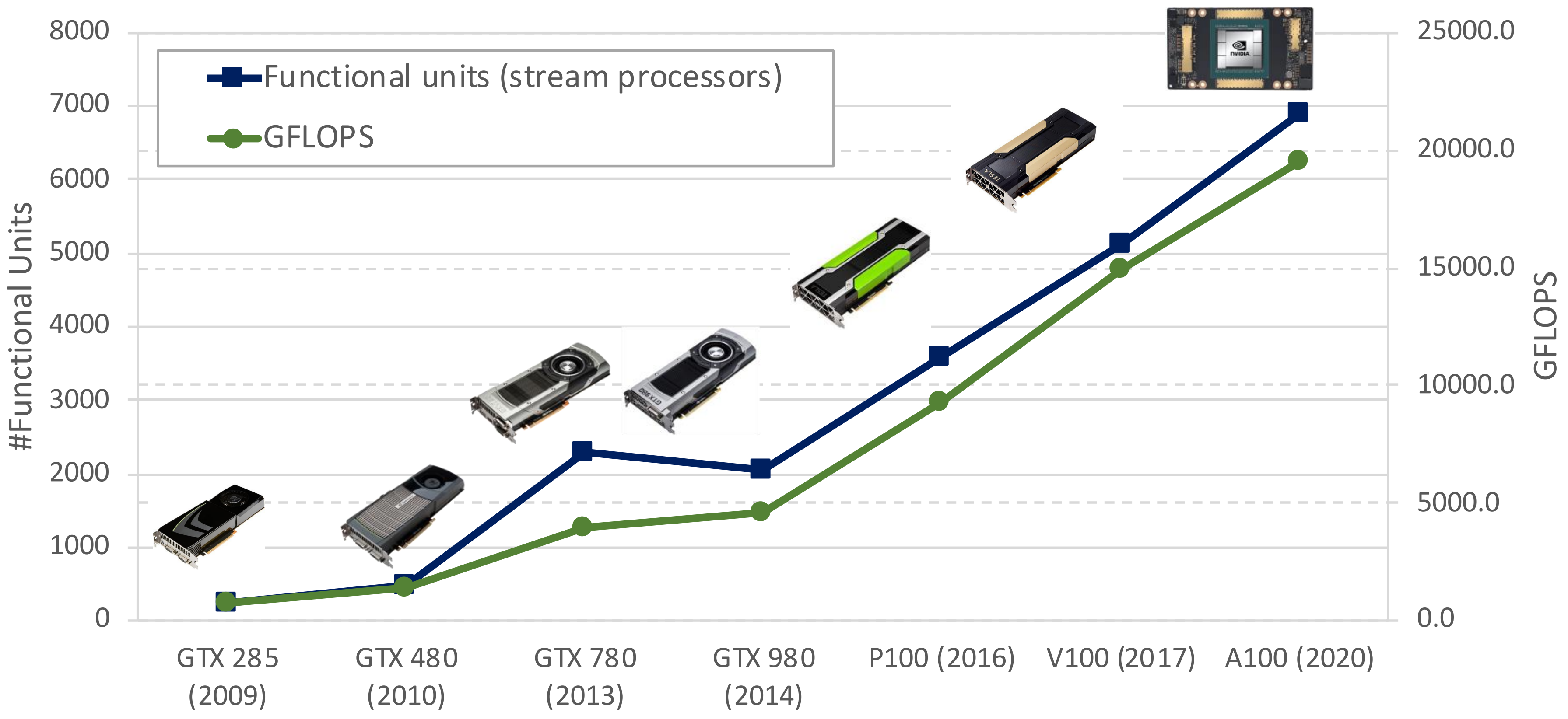
- Groups of 32 threads share instruction stream (each group is a Warp)
- Up to 32 warps are simultaneously interleaved
- Up to 1024 thread contexts can be stored

NVIDIA GeForce GTX 285



30 cores on the GTX 285: 30,720 threads

Evolution of NVIDIA GPUs



NVIDIA V100

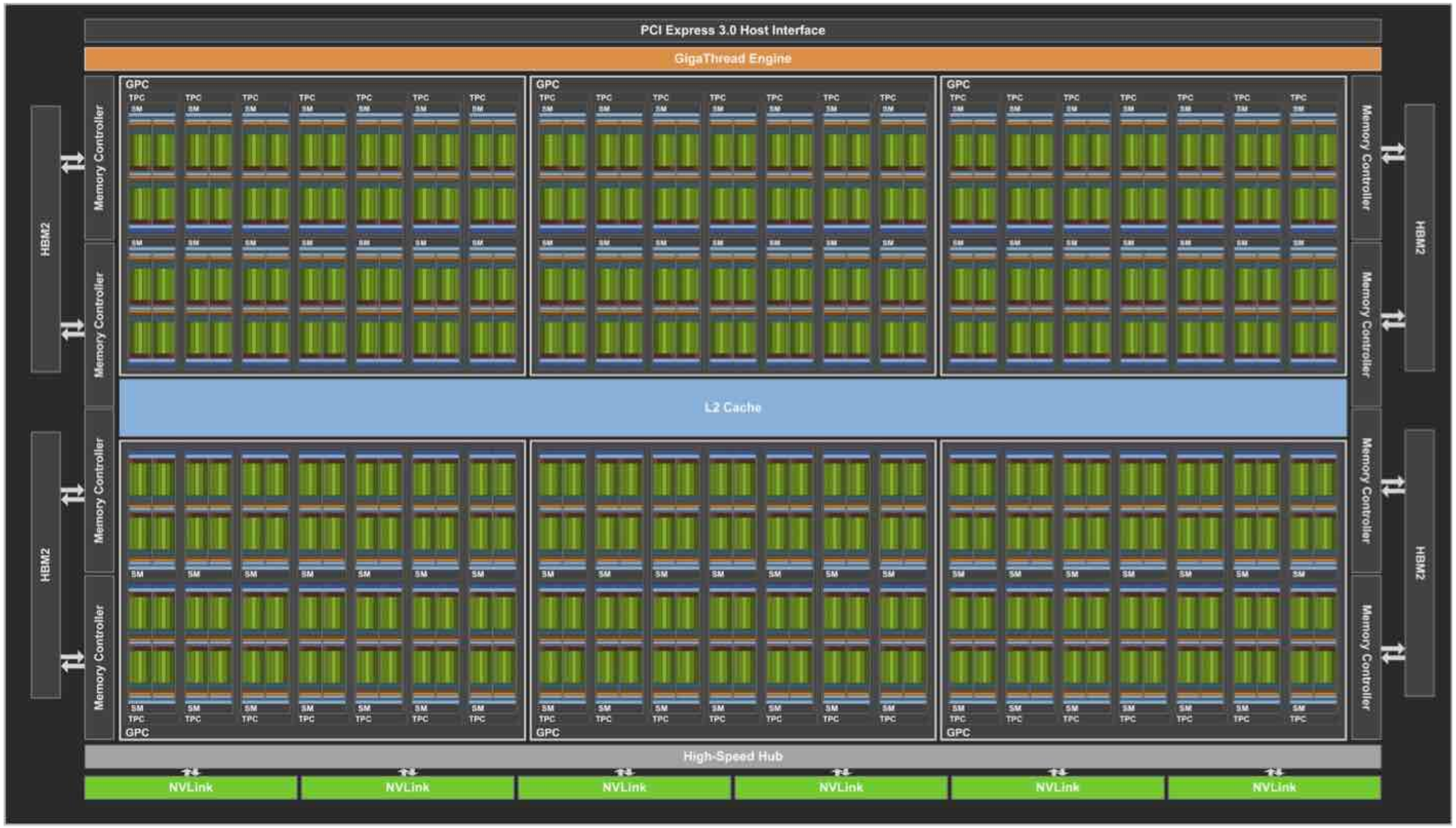
- NVIDIA-speak:
  - 5120 stream processors
  - “SIMT execution”

- Generic speak:
  - 80 cores
  - 64 SIMD functional units per core

  - Tensor cores for Machine Learning

- NVIDIA, “NVIDIA Tesla V100 GPU Architecture. White Paper,” 2017.

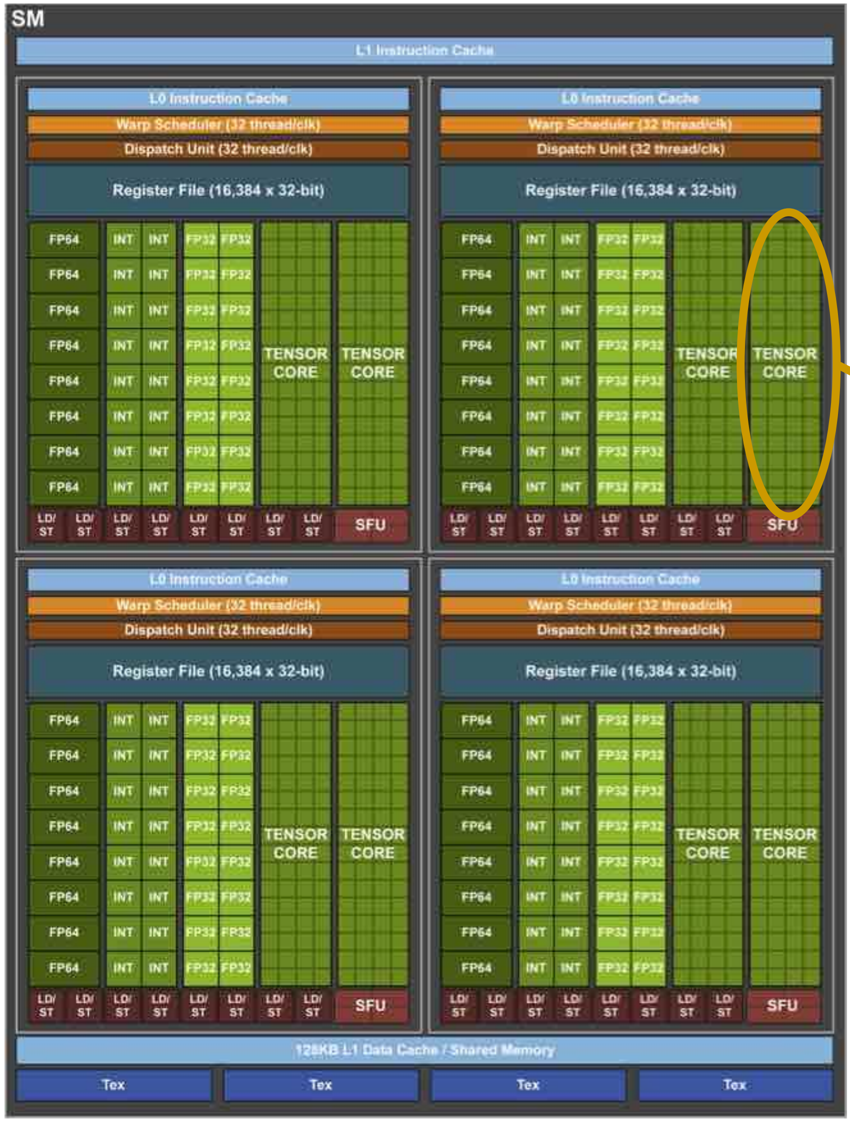
NVIDIA V100 Block Diagram



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

80 cores on the V100

NVIDIA V100 Core

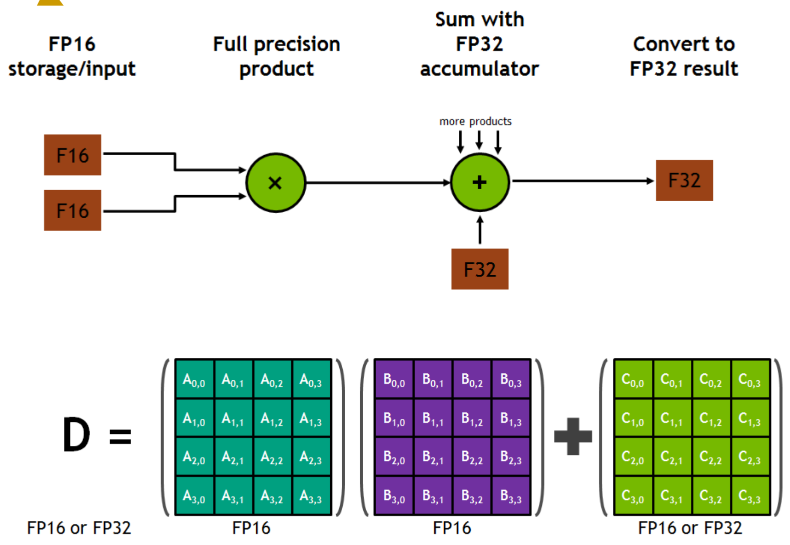


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

15.7 TFLOPS Single Precision

7.8 TFLOPS Double Precision

125 TFLOPS for Deep Learning (Tensor cores)

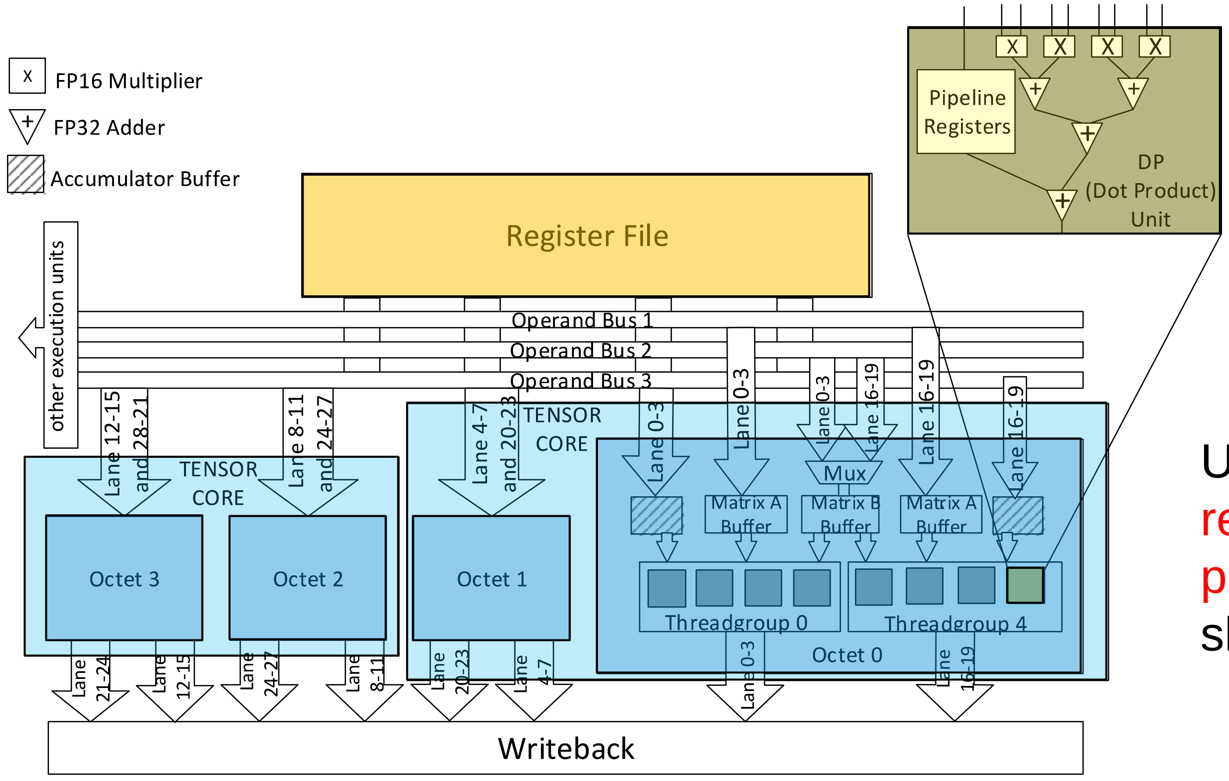


$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0}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

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



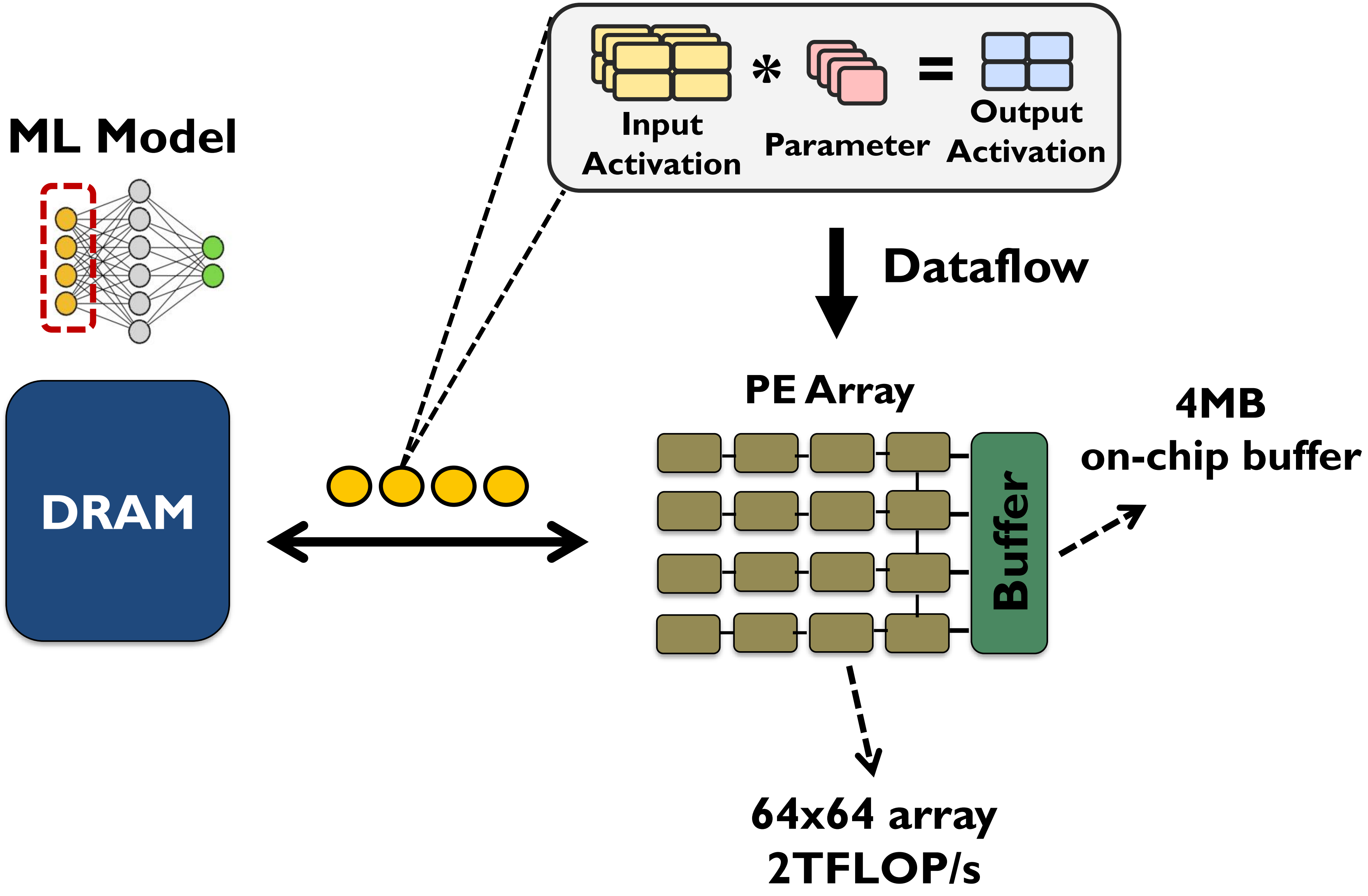
SIMD unit

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

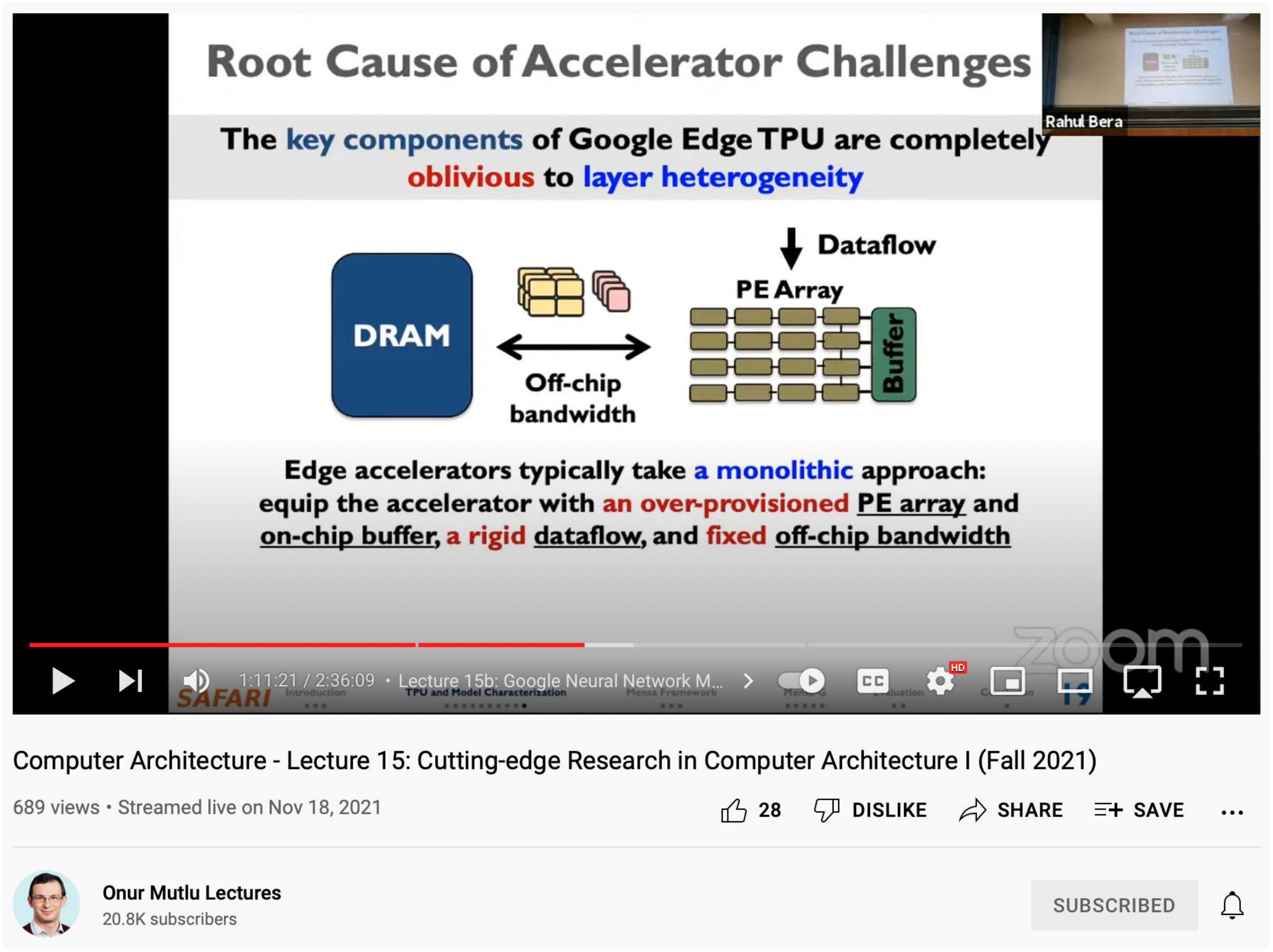
Proposed* tensor core microarchitecture

* 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



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

Computer Architecture - Lecture 15: Cutting-edge Research in Computer Architecture I (Fall 2021)

689 views • Streamed live on Nov 18, 2021

28 DISLIKE SHARE SAVE

Onur Mutlu Lectures
20.8K subscribers

SUBSCRIBED

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

Lecture on Systolic Arrays

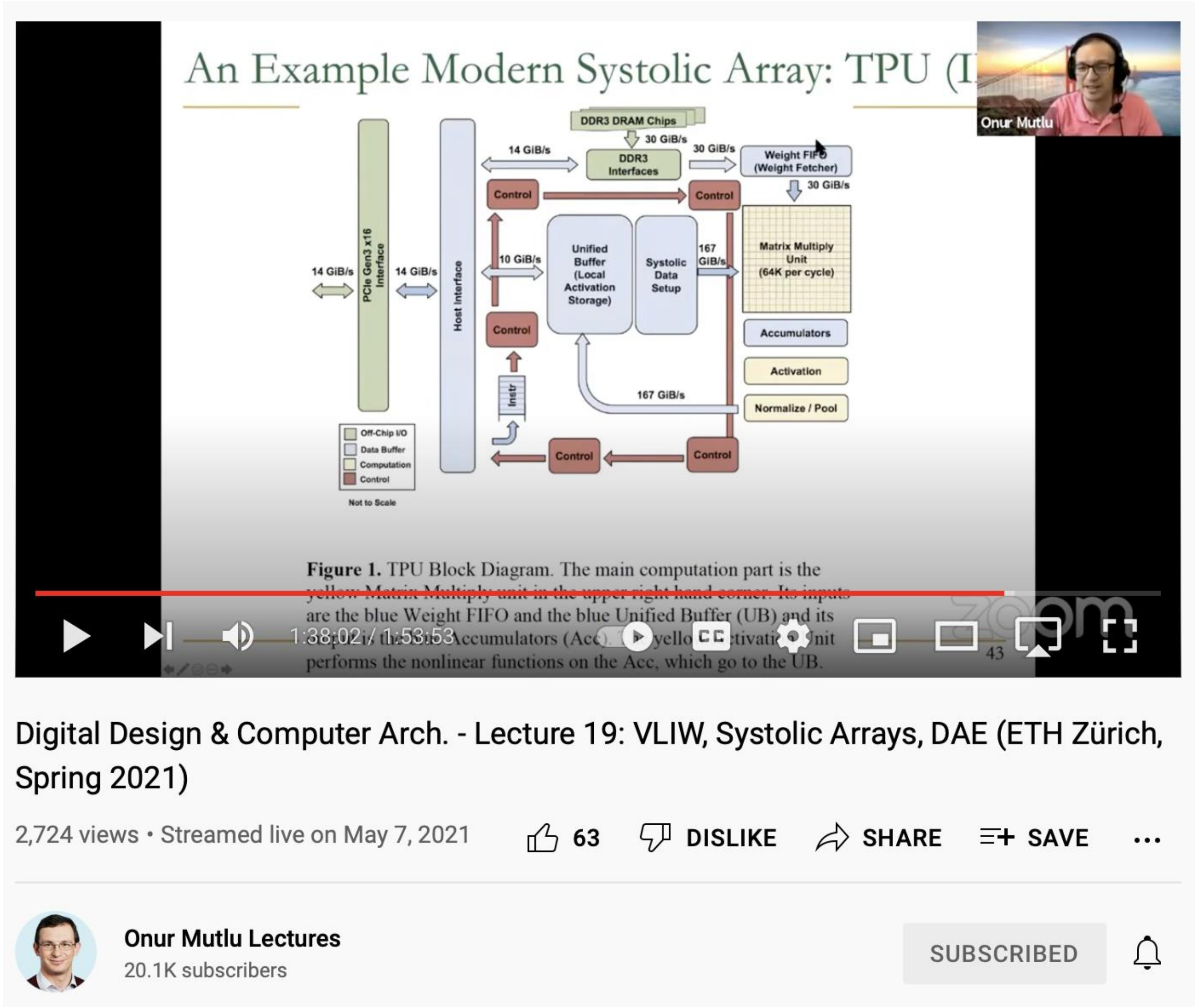


Figure 1. TPU Block Diagram. The main computation part is the yellow Matrix Multiply unit in the upper right hand corner. Its inputs are the blue Weight FIFO and the blue Unified Buffer (UB) and its output is the blue Accumulators (Acc). The yellow Activation Unit performs the nonlinear functions on the Acc, which go to the UB.

Digital Design & Computer Arch. - Lecture 19: VLIW, Systolic Arrays, DAE (ETH Zürich, Spring 2021)

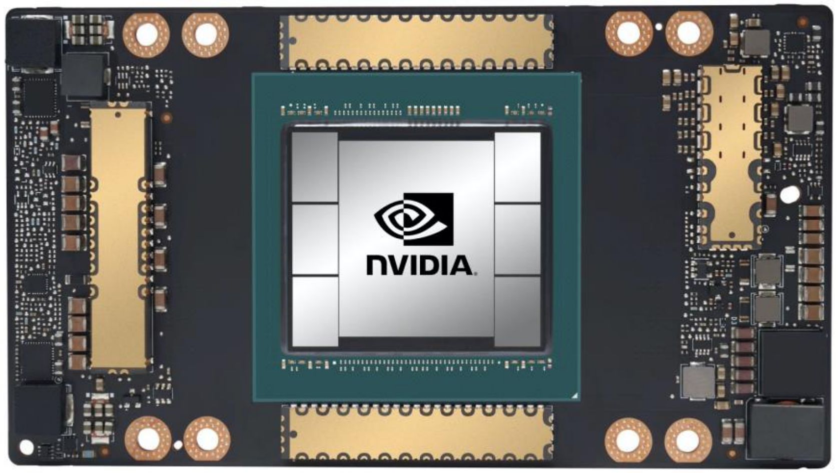
2,724 views • Streamed live on May 7, 2021 63 DISLIKE SHARE SAVE ...

Onur Mutlu Lectures
20.1K subscribers
SUBSCRIBED

https://youtu.be/UtLy4Yagdys?t=2948

NVIDIA A100

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

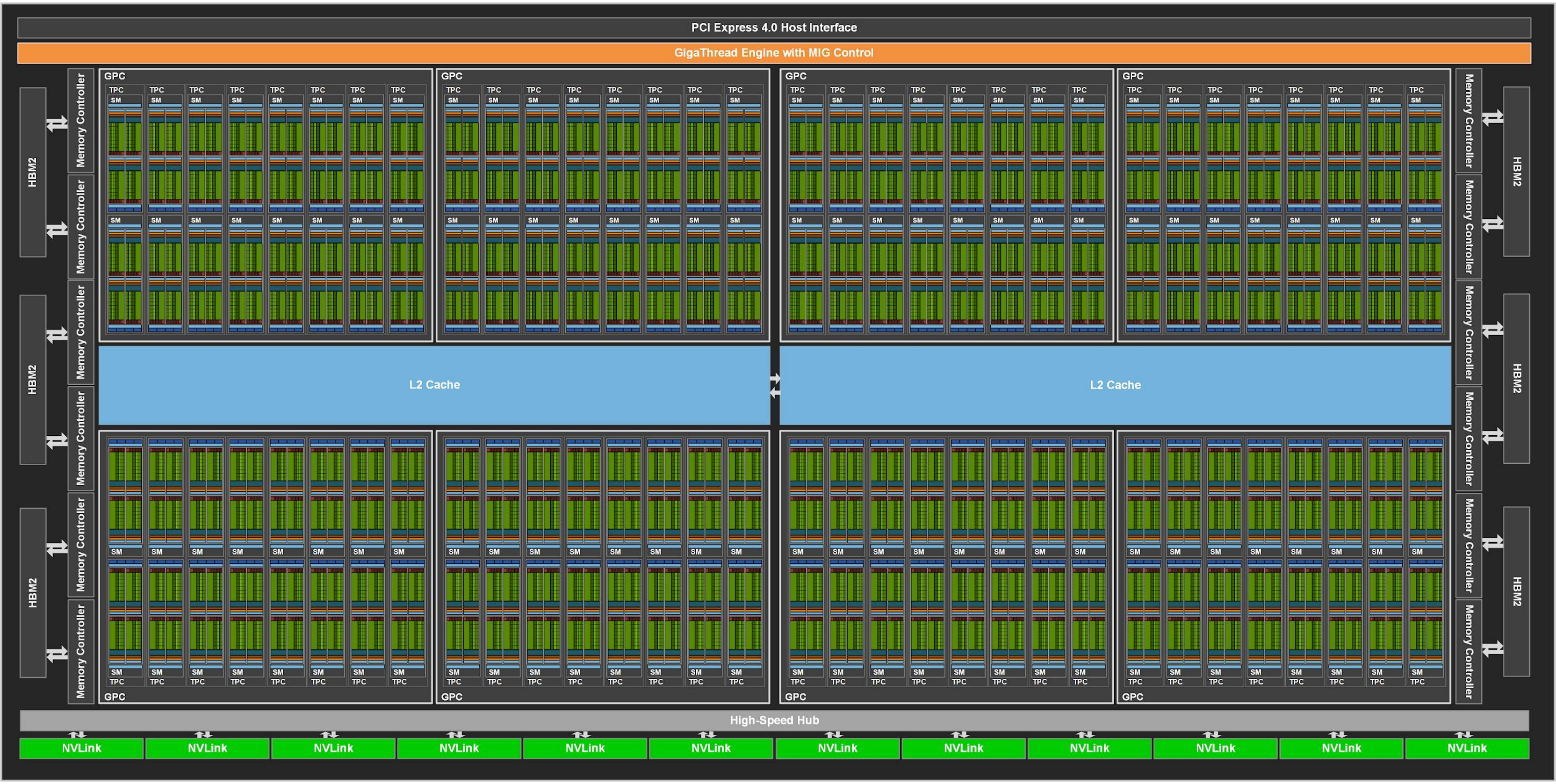


- Generic speak:
  - 108 cores
  - 64 SIMD functional units per core

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

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

NVIDIA A100 Block Diagram



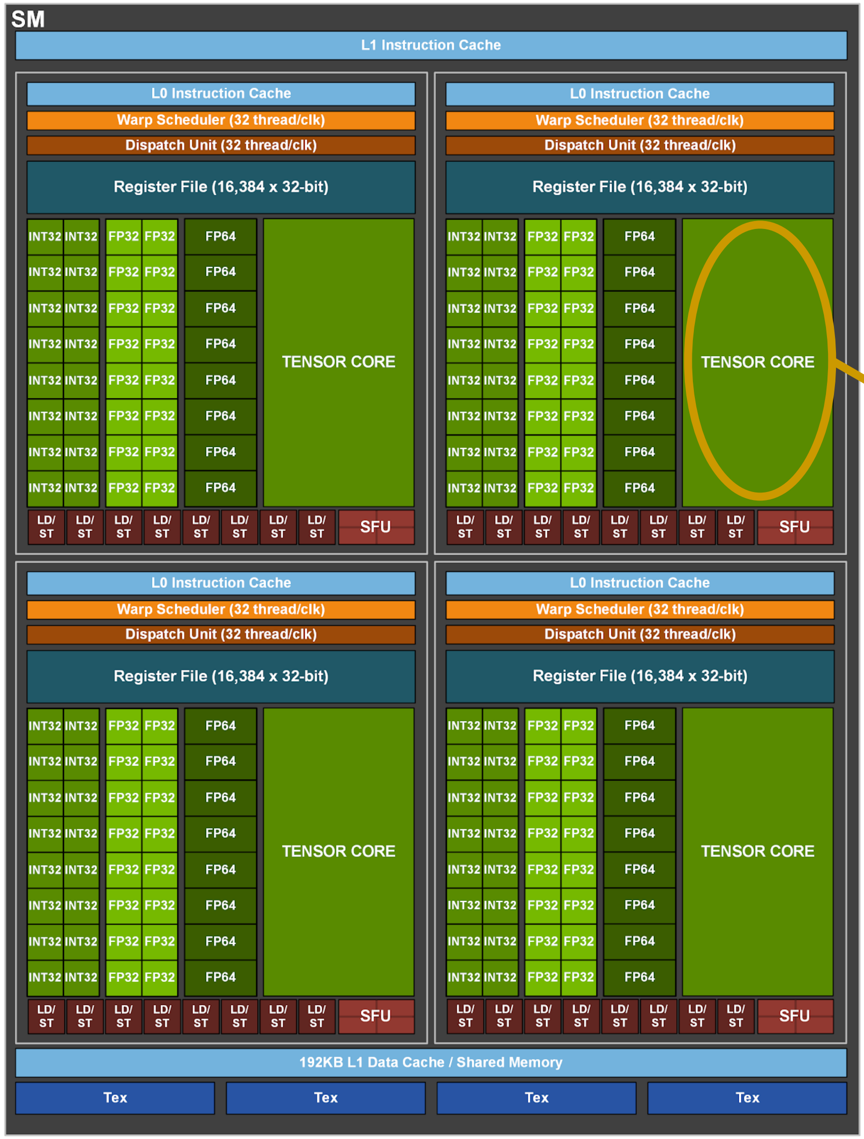
https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

108 cores on the A100

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

40MB L2 cache

NVIDIA A100 Core

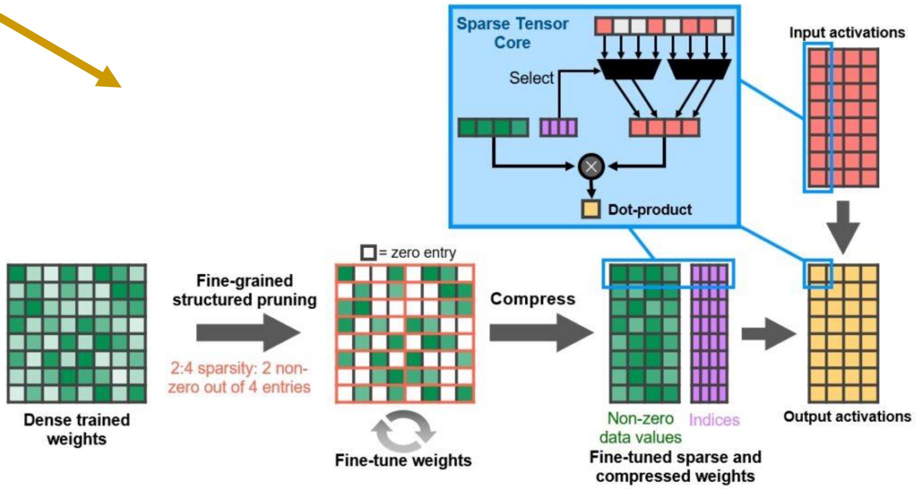


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

19.5 TFLOPS Single Precision

9.7 TFLOPS Double Precision

312 TFLOPS for Deep Learning (Tensor cores)



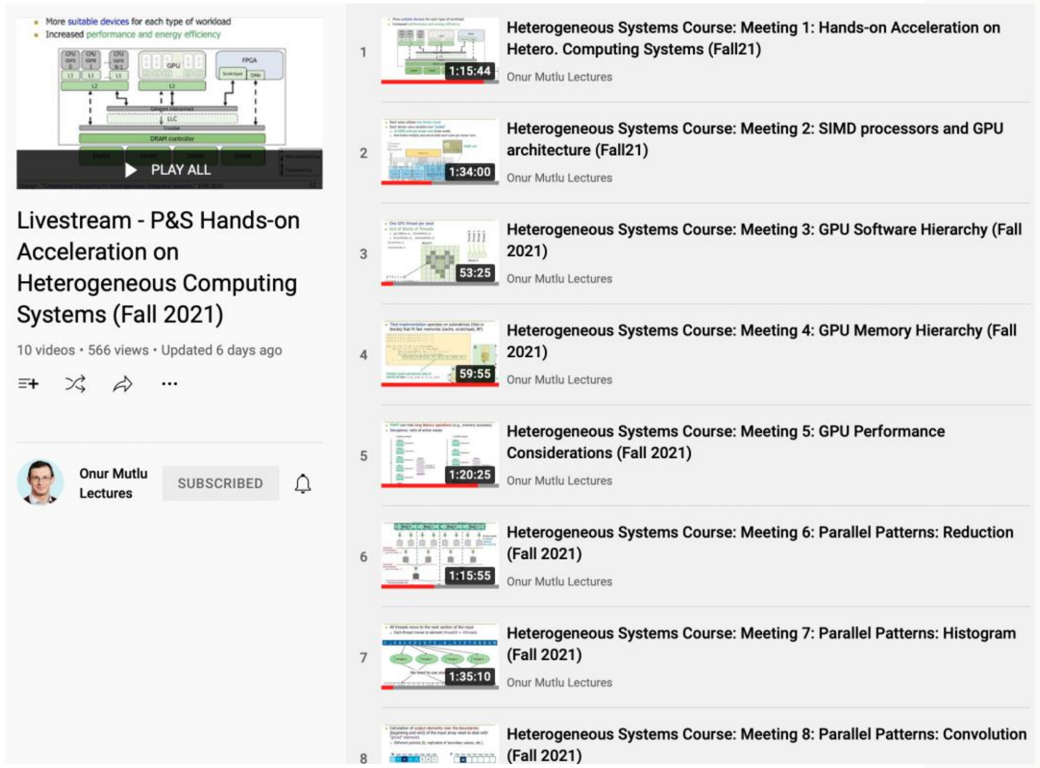
https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

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?

Heterogeneous Systems Course (Fall 2021)

- Short weekly lectures
- Hands-on projects



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

10 videos • 566 views • Updated 6 days ago

1 Heterogeneous Systems Course: Meeting 1: Hands-on Acceleration on Hetero. Computing Systems (Fall21) — 1:15:44 — Onur Mutlu Lectures

2 Heterogeneous Systems Course: Meeting 2: SIMD processors and GPU architecture (Fall21) — 1:34:00 — Onur Mutlu Lectures

3 Heterogeneous Systems Course: Meeting 3: GPU Software Hierarchy (Fall 2021) — 53:25 — Onur Mutlu Lectures

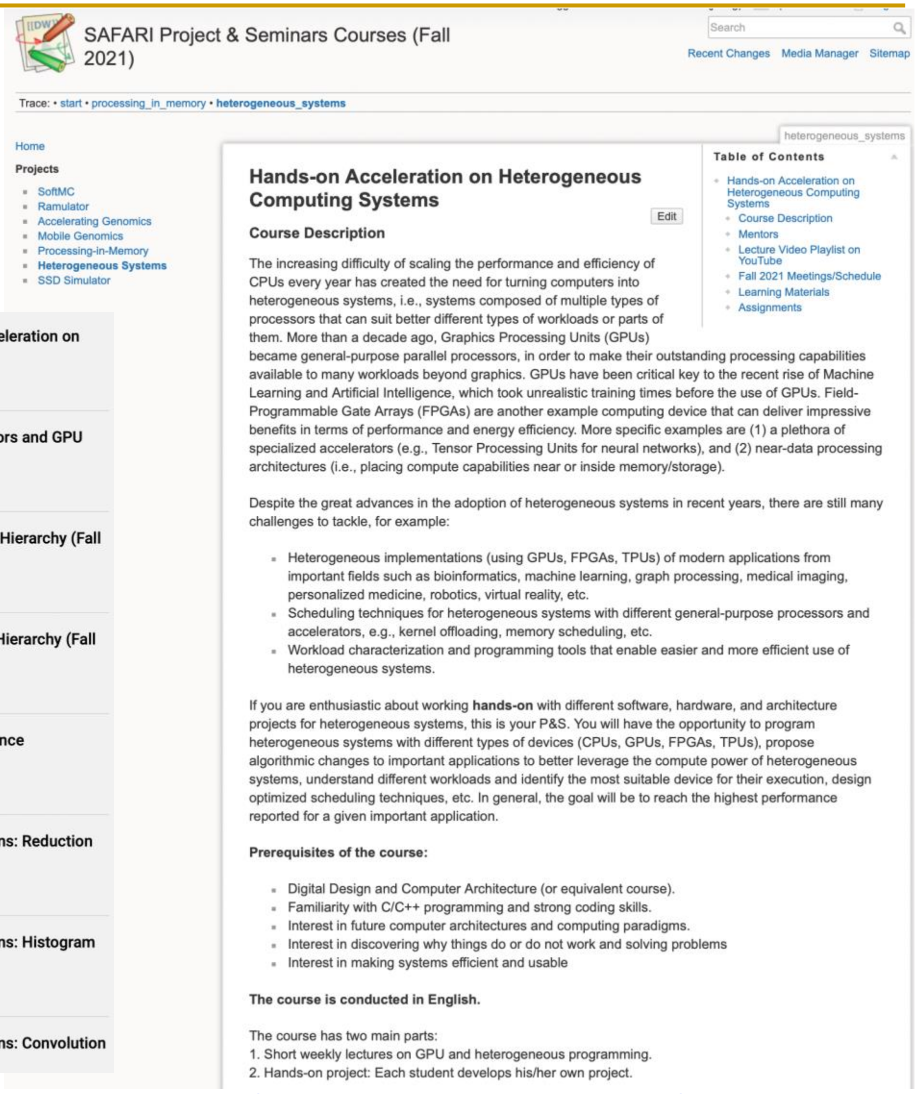
4 Heterogeneous Systems Course: Meeting 4: GPU Memory Hierarchy (Fall 2021) — 59:55 — Onur Mutlu Lectures

5 Heterogeneous Systems Course: Meeting 5: GPU Performance Considerations (Fall 2021) — 1:20:25 — Onur Mutlu Lectures

6 Heterogeneous Systems Course: Meeting 6: Parallel Patterns: Reduction (Fall 2021) — 1:15:55 — Onur Mutlu Lectures

7 Heterogeneous Systems Course: Meeting 7: Parallel Patterns: Histogram (Fall 2021) — 1:35:10 — Onur Mutlu Lectures

8 Heterogeneous Systems Course: Meeting 8: Parallel Patterns: Convolution (Fall 2021)



SAFARI Project & Seminars Courses (Fall 2021)

Trace: • start • processing_in_memory • heterogeneous_systems

Home

Projects
- SoftMC
- Ramulator
- Accelerating Genomics
- Mobile Genomics
- Processing-in-Memory
- Heterogeneous Systems
- SSD Simulator

Table of Contents
- Hands-on Acceleration on Heterogeneous Computing Systems
  - Course Description
  - Mentors
  - Lecture Video Playlist on YouTube
  - Fall 2021 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=PL5Q2soXY2Zi_OwkTgEyA6tk3UsoPBH737

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

Computer Architecture
Lecture 25: SIMD Processors and GPUs

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich

Fall 2022

05 January 2023

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