A Brief Overview of SIMD Architectures: Vector, SIMD Extensions and GPUs

Slides adapted from Onur Mutlu (CMU)

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

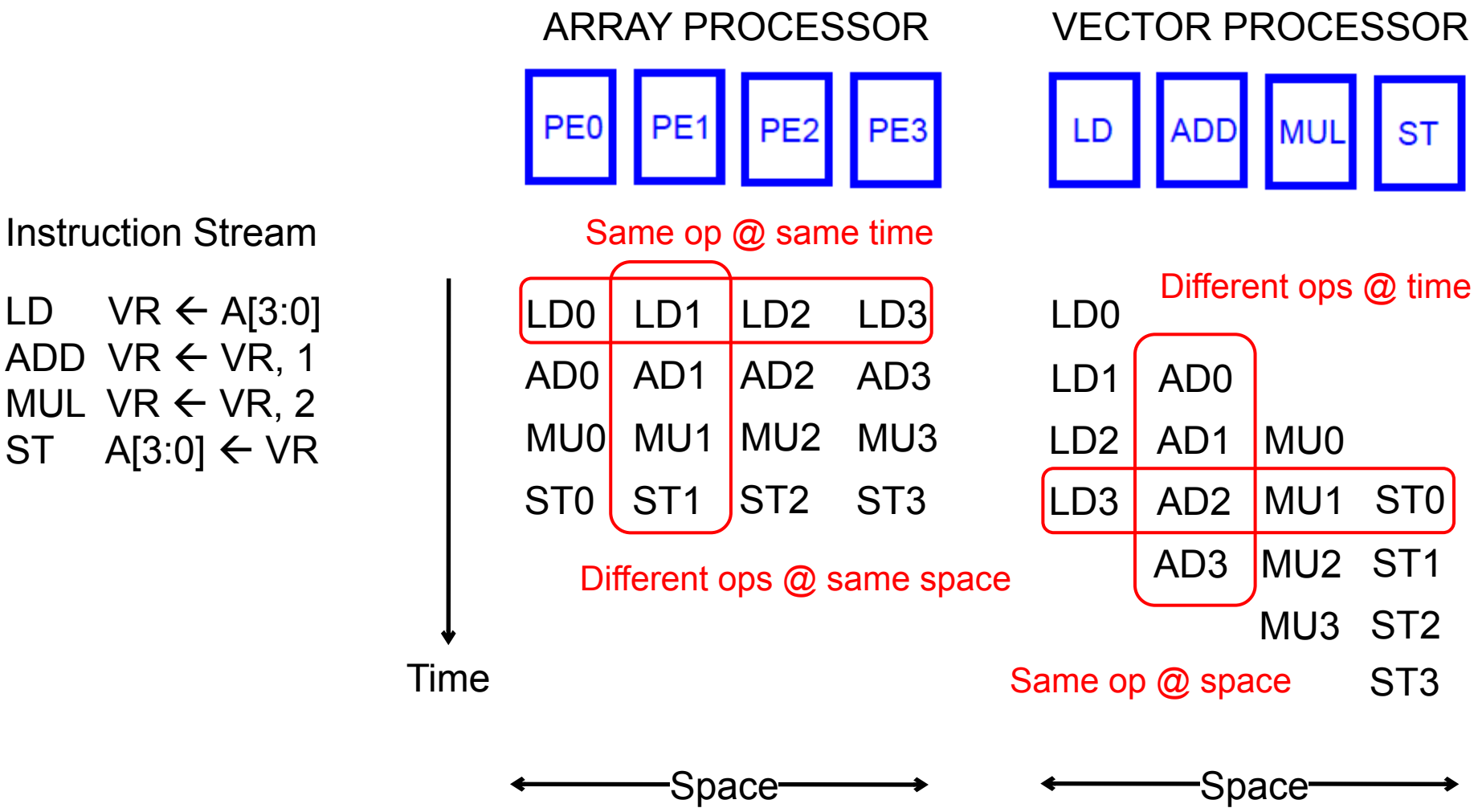
- Concurrency arises from performing the same operations on different pieces of data
  - Single instruction multiple data (SIMD)
  - E.g., dot product of two vectors
- Contrast with data flow
  - Concurrency arises from executing different operations in parallel (in a data driven manner)
- Contrast with thread ("control") parallelism
  - Concurrency arises from executing different threads of control in parallel
- SIMD exploits instruction-level parallelism
  - Multiple instructions concurrent: instructions happen to be the same

SIMD Processing

- Single instruction operates on multiple data elements
  - In time or in space
- Multiple processing elements

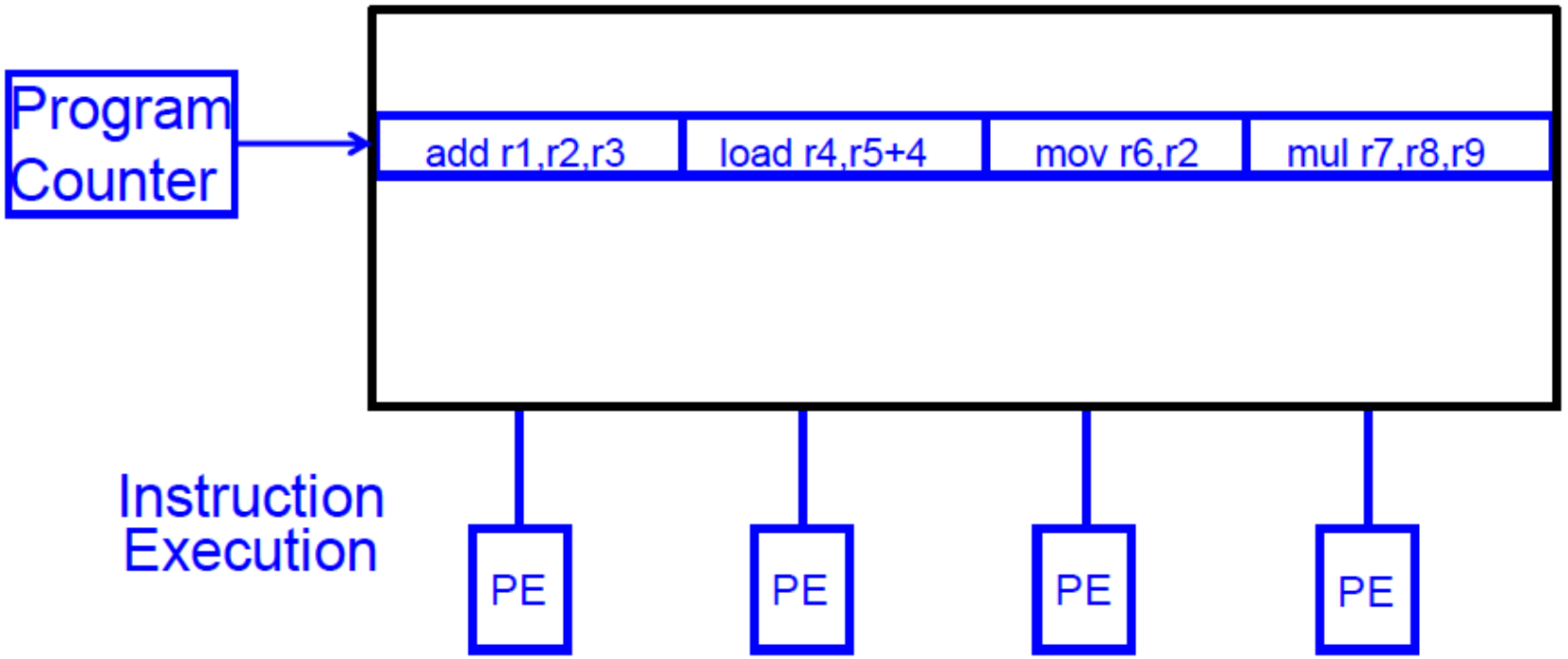
- Time-space duality
  - Array processor: Instruction operates on multiple data elements at the same time
  - Vector processor: Instruction operates on multiple data elements in consecutive time steps

Array vs. Vector Processors



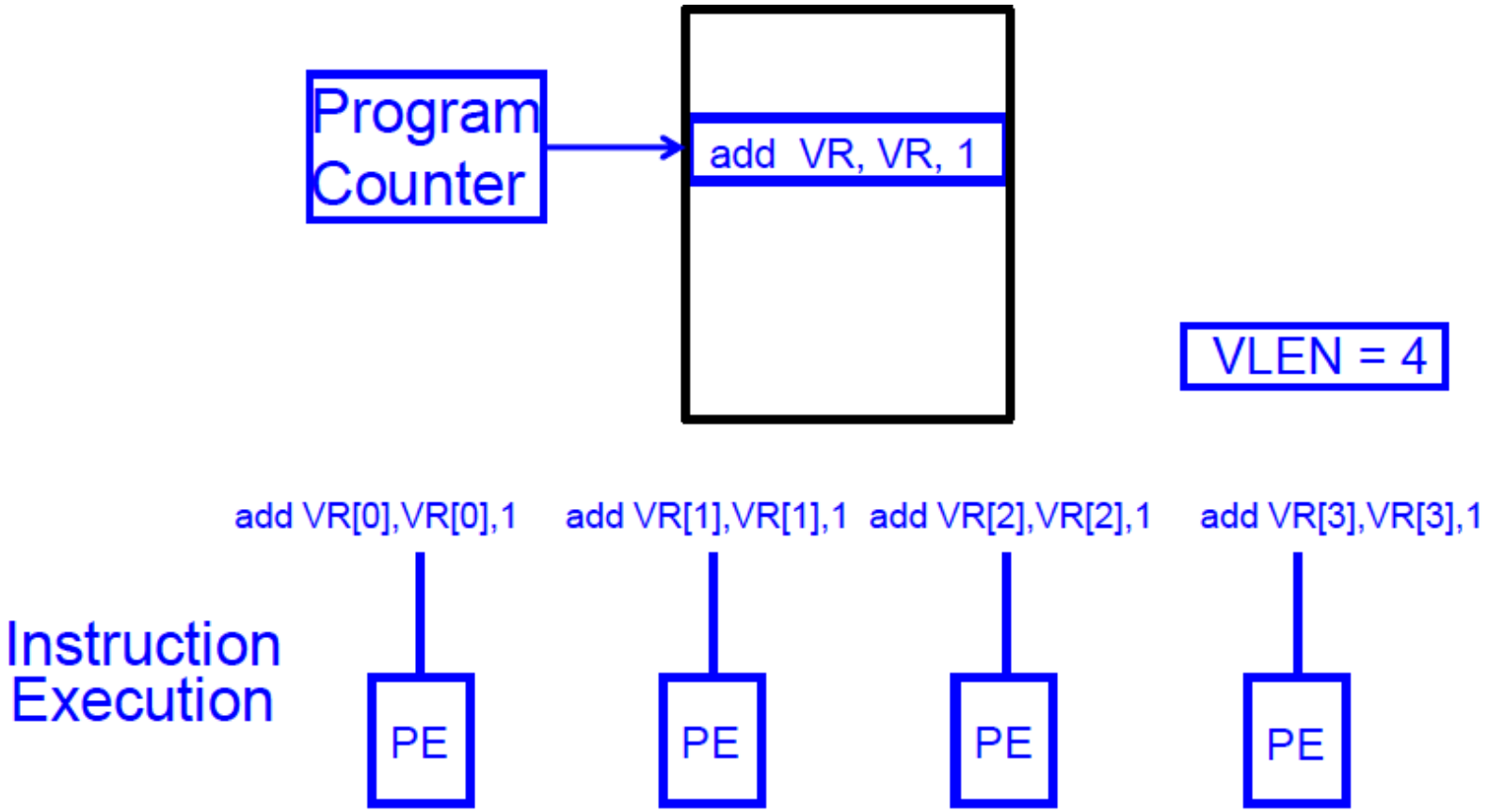
SIMD Array Processing vs. VLIW

- VLIW



SIMD Array Processing vs. VLIW

- Array processor



Vector Processors

- 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 between two elements of a vector

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 within a vector
  - No control flow within a vector
  - Known stride allows prefetching of vectors into cache/memory

Vector Processor Advantages

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

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

+ Highly regular memory access pattern
  - Interleaving multiple banks for higher memory bandwidth
  - Prefetching

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

Vector Processor Disadvantages

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

++ Vector operations

-- Very inefficient if parallelism is irregular

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

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

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

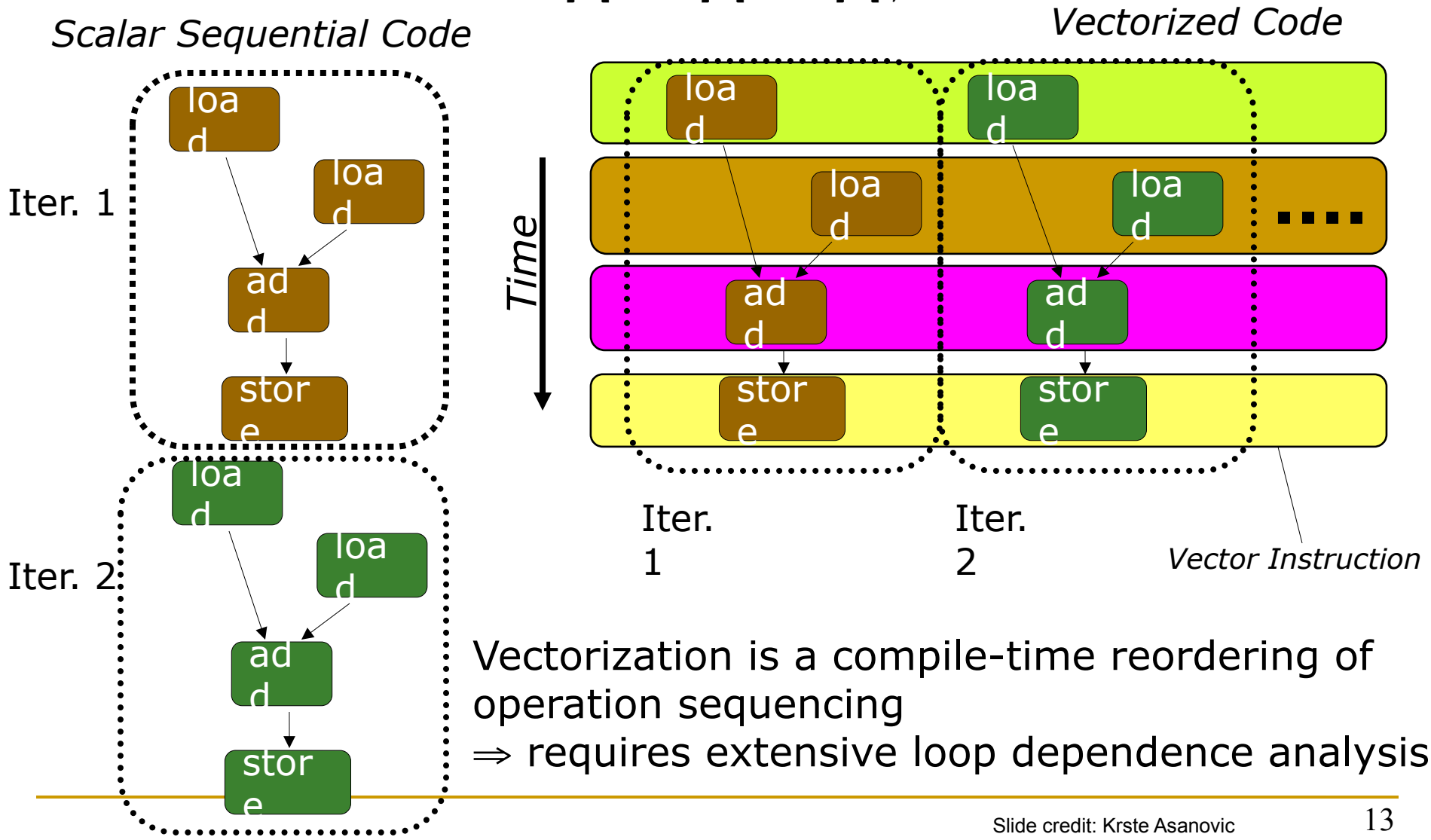
Vector Processor Limitations

-- 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

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

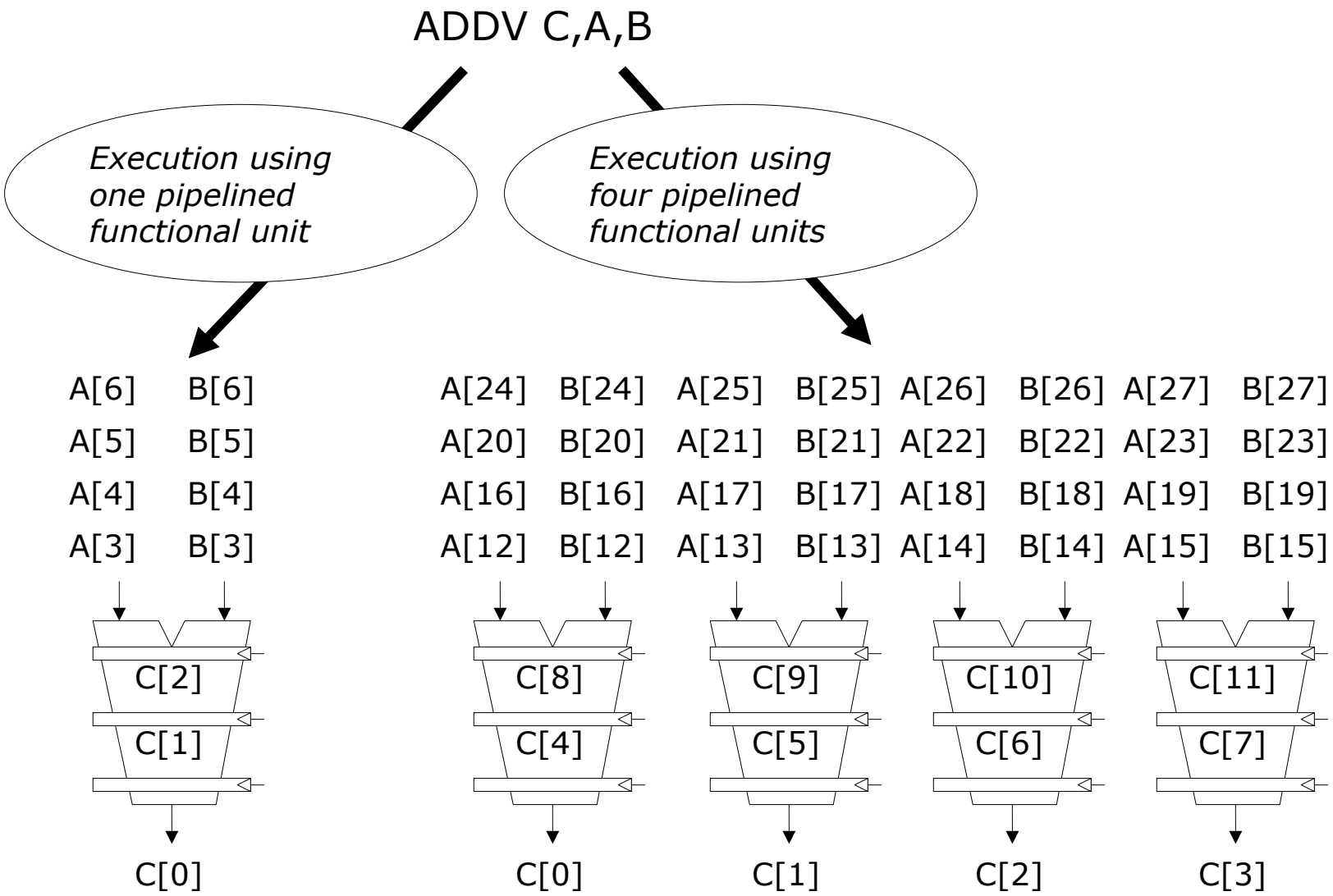
Vector/SIMD Processing Summary

- Vector/SIMD machines good at exploiting regular data-level parallelism
  - Same operation performed on many data elements
  - Improve performance, simplify design (no intra-vector dependencies)

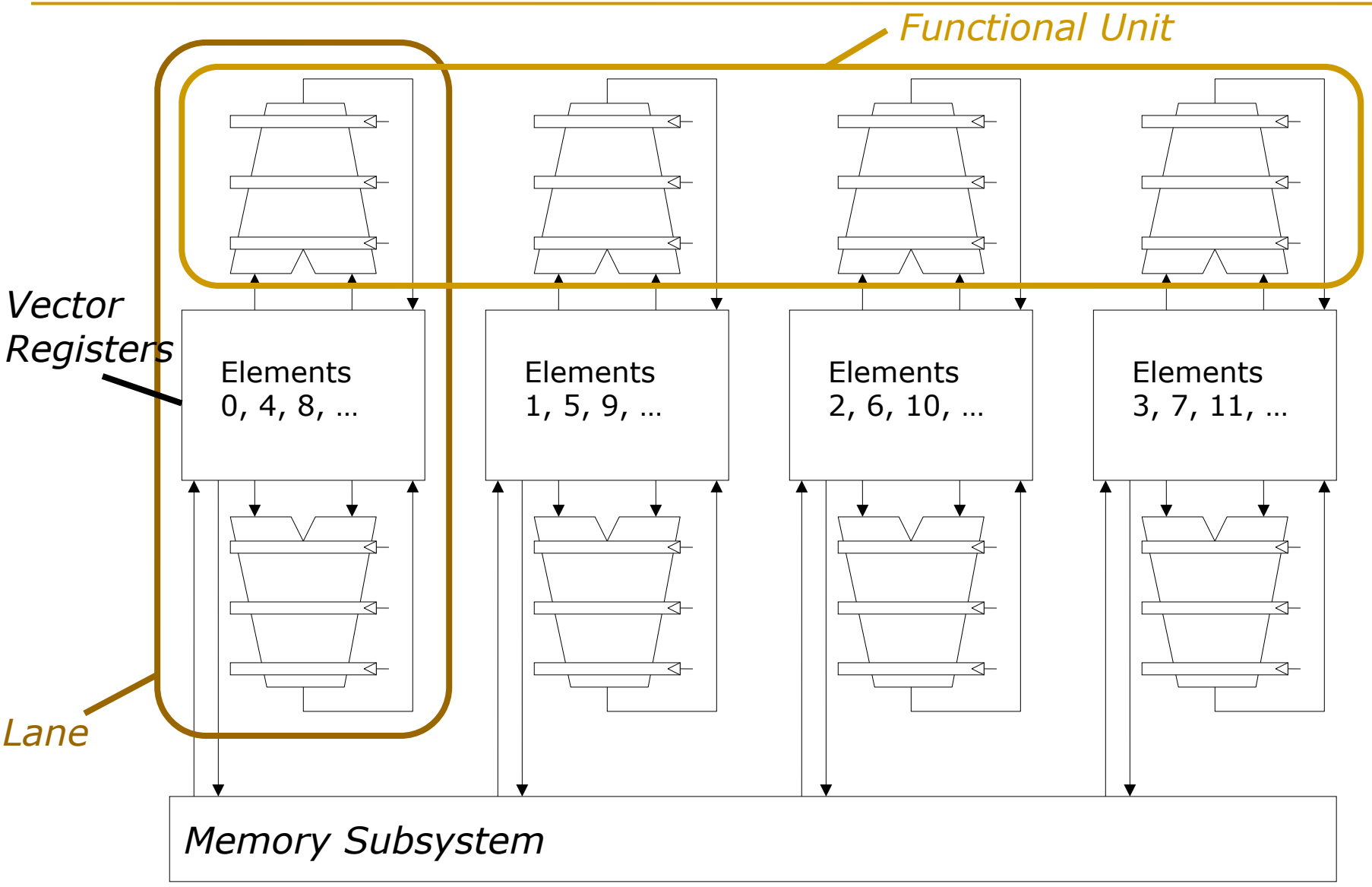
- Performance improvement limited by vectorizability of code
  - Scalar operations limit vector machine performance
  - Amdahl's Law
  - CRAY-1 was the fastest SCALAR machine at its time!

- Many existing ISAs include (vector-like) SIMD operations
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD

Vector Instruction Execution



Vector Unit Structure

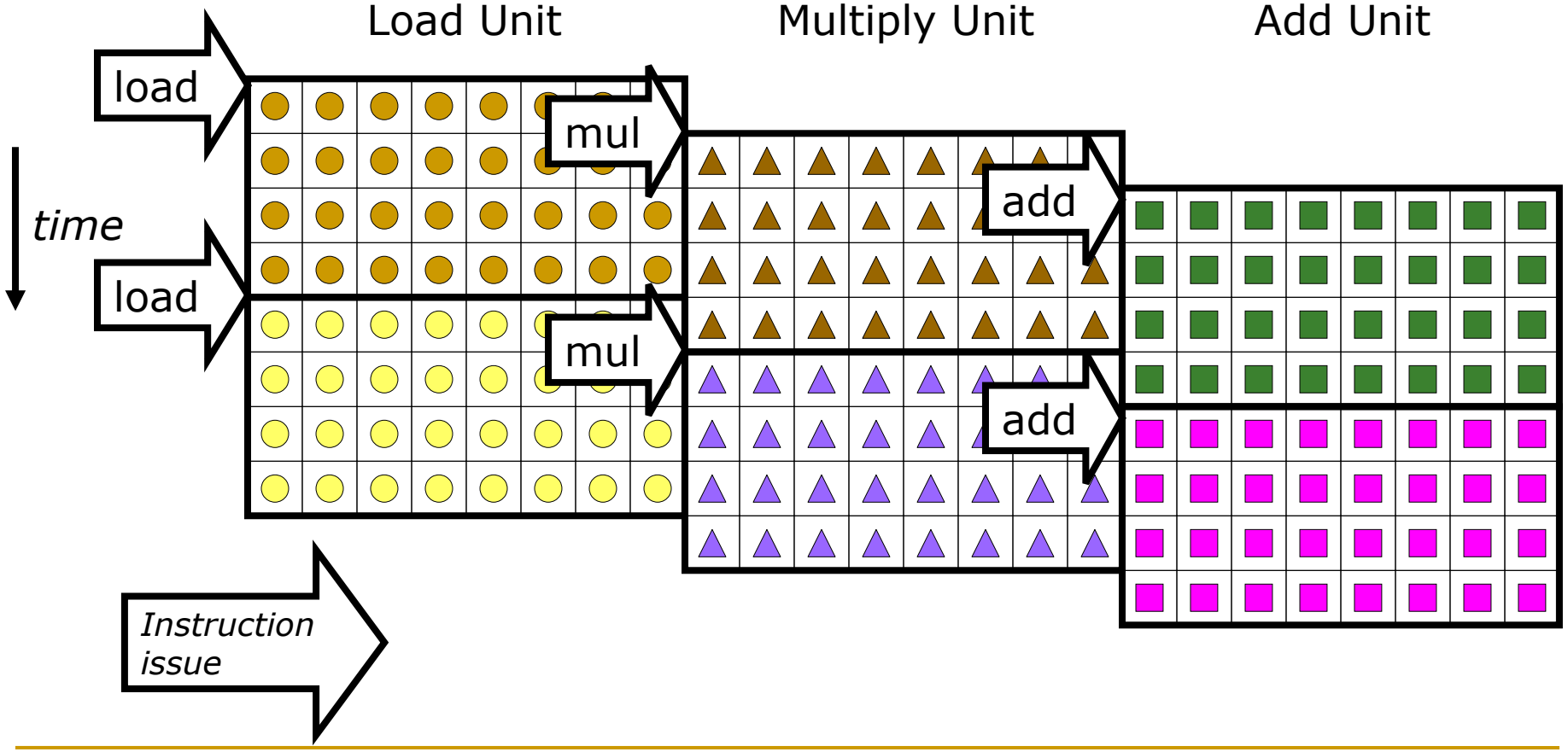


Slide credit: Krste Asanovic

Vector Instruction Level Parallelism

Can overlap execution of multiple vector instructions

- example machine has 32 elements per vector register and 8 lanes
- Complete 24 operations/cycle while issuing 1 short instruction/cycle



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

SIMD Operations in Modern ISAs

Intel Pentium MMX Operations

- Idea: One instruction operates on multiple data elements simultaneously
  - Designed with multimedia (graphics) operations in mind

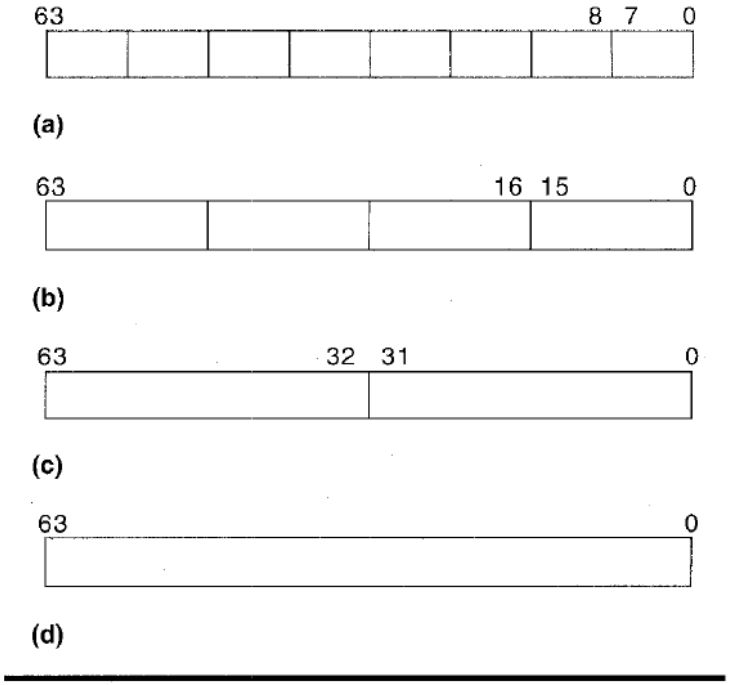


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

No VLEN register
Opcode determines data type:
8 8-bit bytes
4 16-bit words
2 32-bit doublewords
1 64-bit quadword

Stride always equal to 1.

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

MMX Example: Image Overlaying

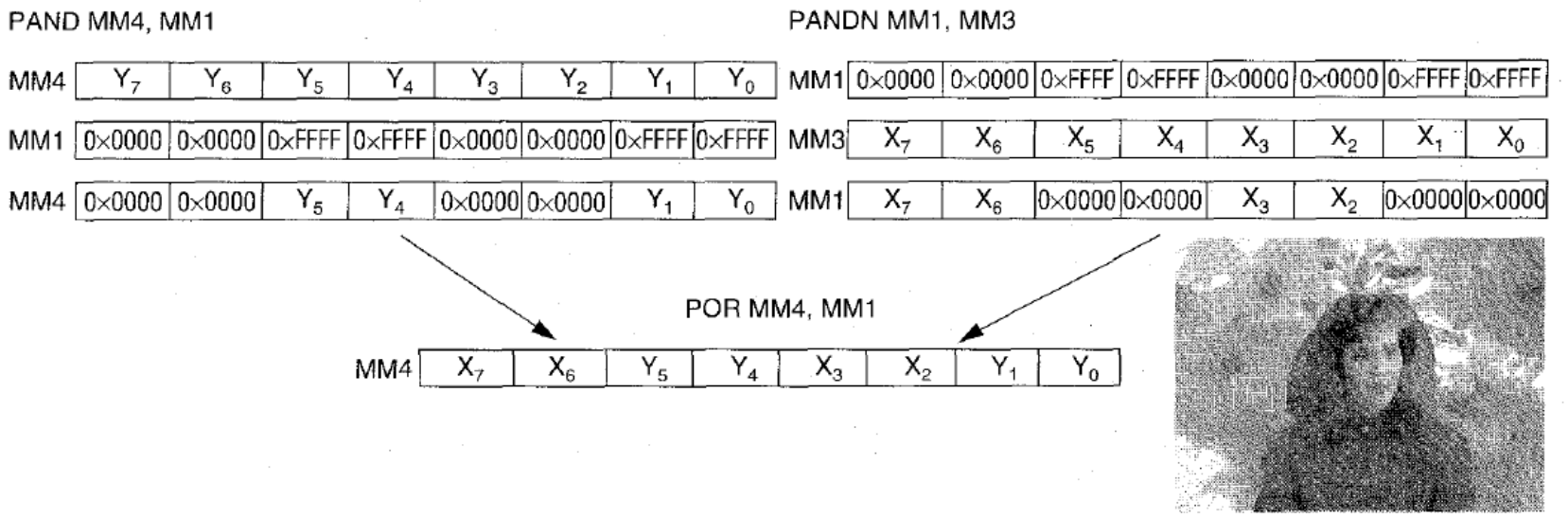


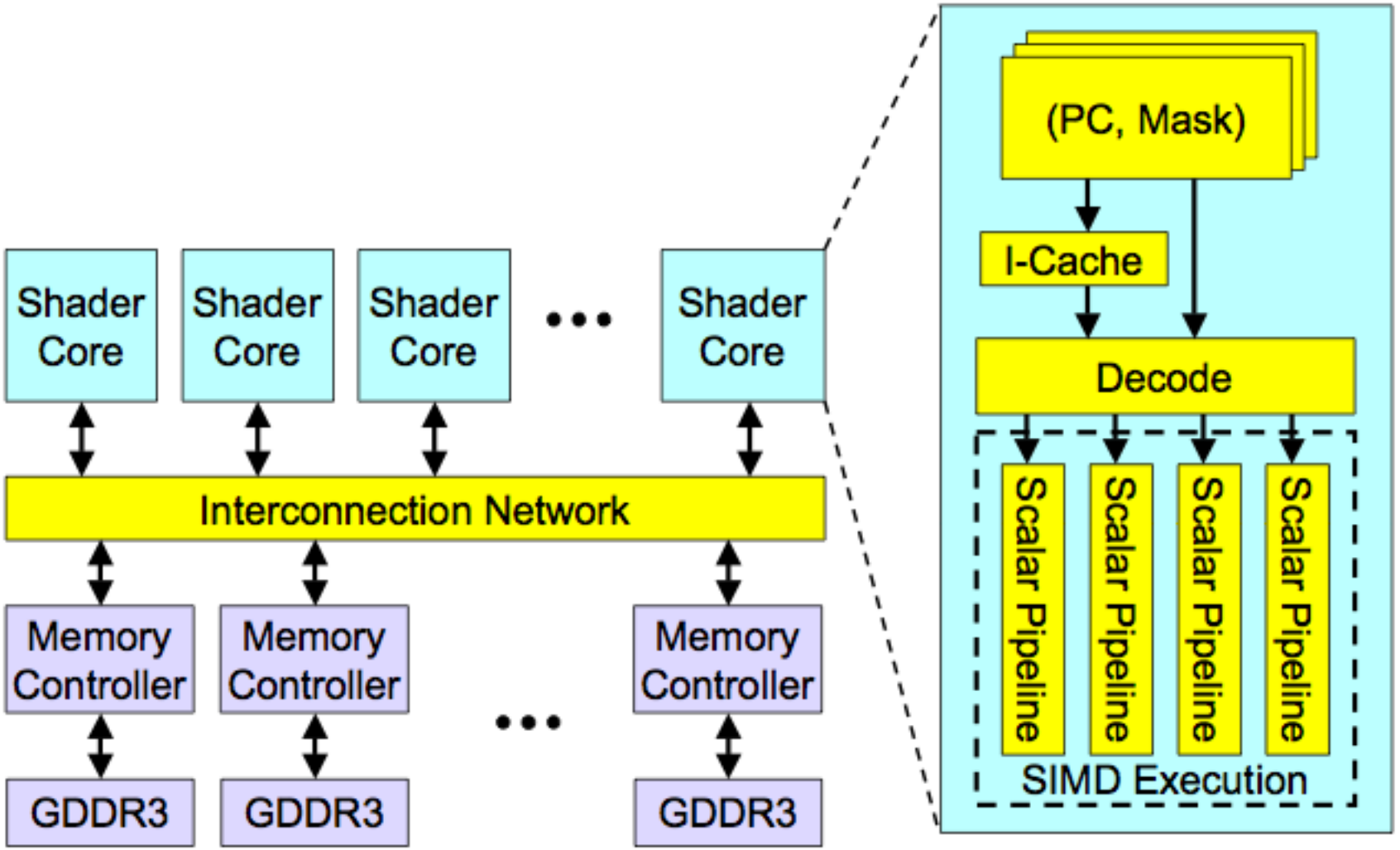
Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

```
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
```

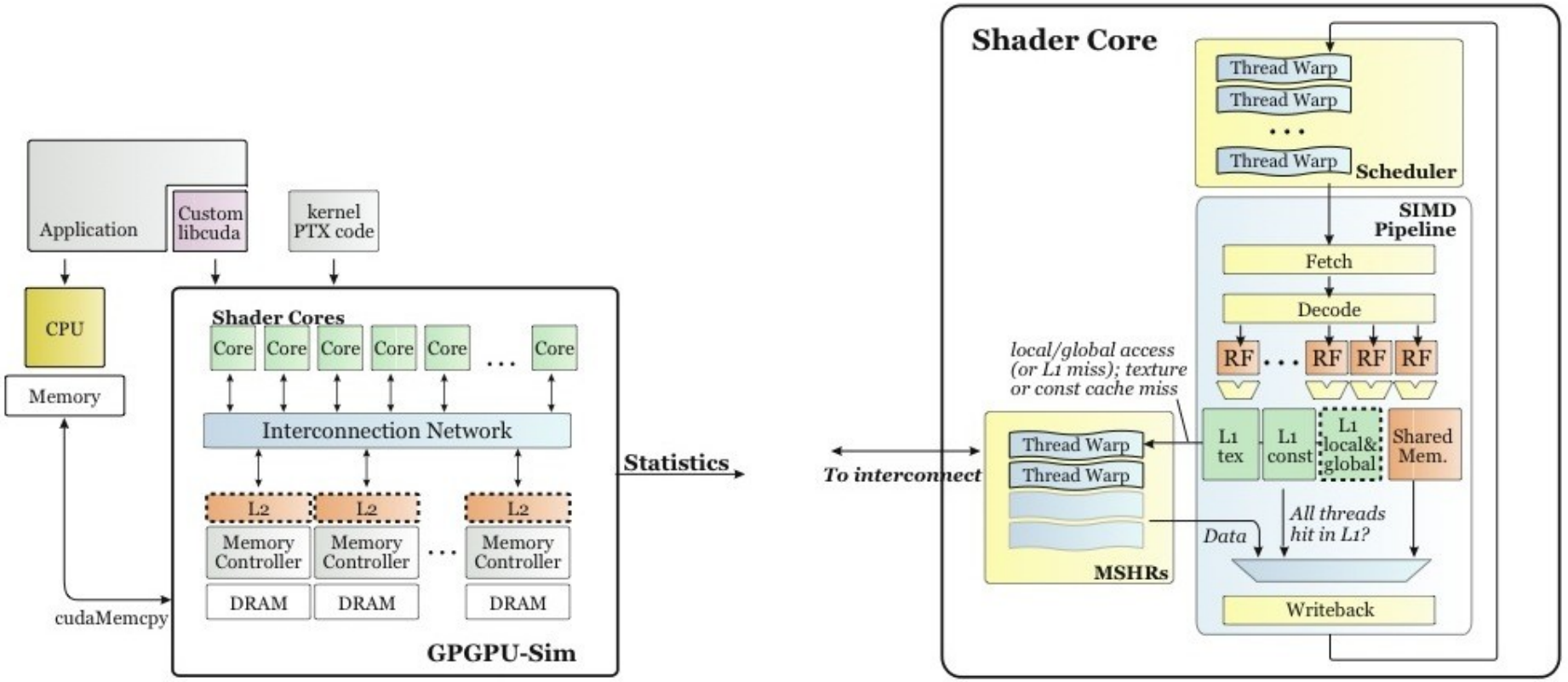
Figure 11. MMX code sequence for performing a conditional select.

Graphics Processing Units
SIMD not Exposed to Programmer (SIMT)

High-Level View of a GPU



Overall Figure - GPGPU



Slide credit: GPGPUSim Manual

CUDA Programming Model

- Kernel is a grid of threads
- Each grid is a group of blocks (or CTAs)
- Each block is a group of warps (= 32 threads)

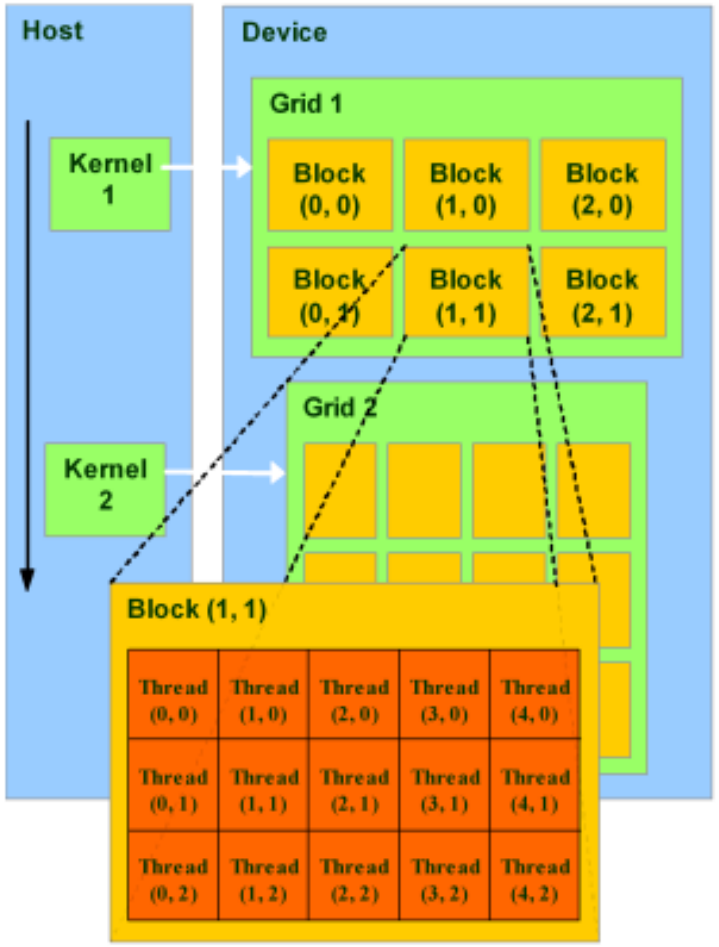
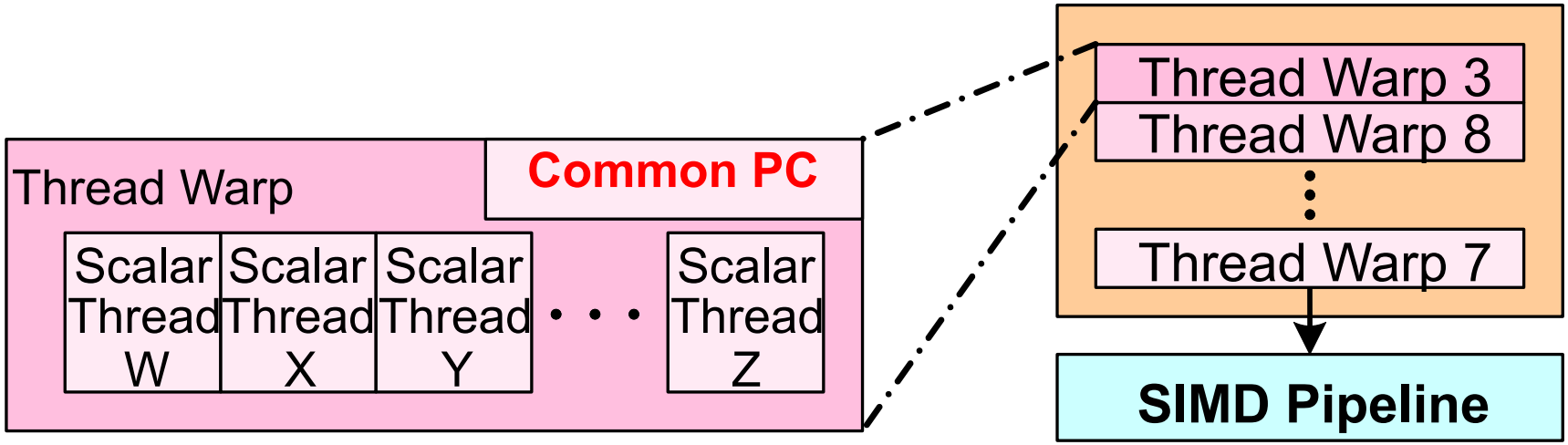


Image credit: http://ixbtlabs.com/

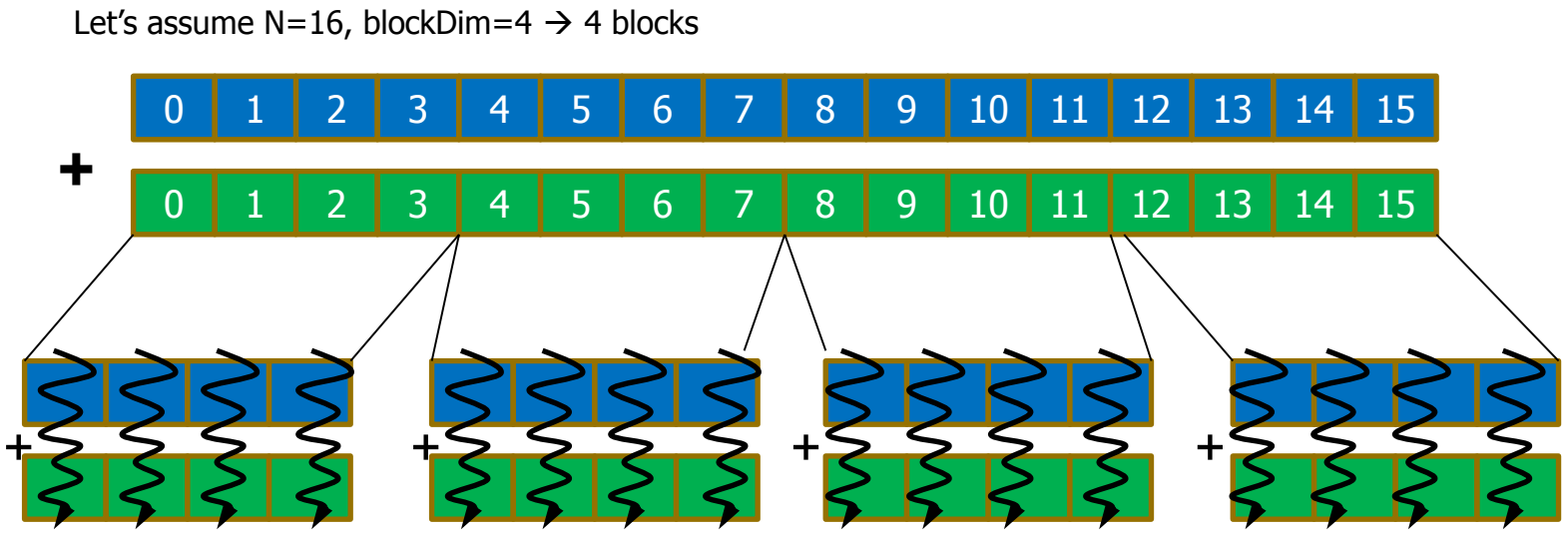
Concept of "Thread Warps" and SIMT

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



SIMT Memory Access

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



Slide credit: Hyesoon Kim

Sample GPU SIMT Code (Simplified)

CPU code

```
for (ii = 0; ii < 100; ++ii) {
C[ii] = A[ii] + B[ii];
}
```

CUDA code

```
// there are 100 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

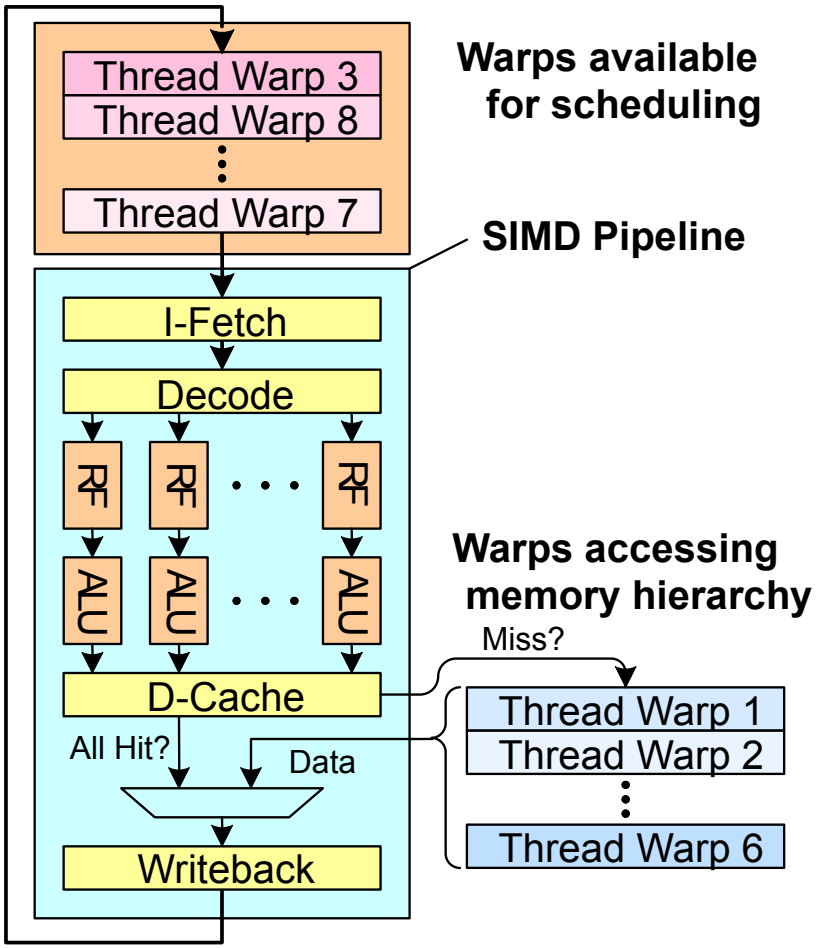
```
__global__  add_matrix
  ( float *a, float *b, float *c, int N) {
int i = blockIdx.x *  blockDim.x + threadIdx.x;
Int j = blockIdx.y * blockDim.y  + threadIdx.y;
int index = i + j*N;
 if (i < N && j < N)
   c[index] = a[index]+b[index];
}

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

Latency Hiding with "Thread Warps"

- 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 branch prediction)
  - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- No OS context switching
- Memory latency hiding
  - Graphics has millions of pixels



Warp-based SIMD vs. Traditional SIMD

- Traditional SIMD contains a single thread
  - Lock step
  - Programming model is SIMD (no threads) → SW needs to know vector length
  - ISA contains vector/SIMD instructions

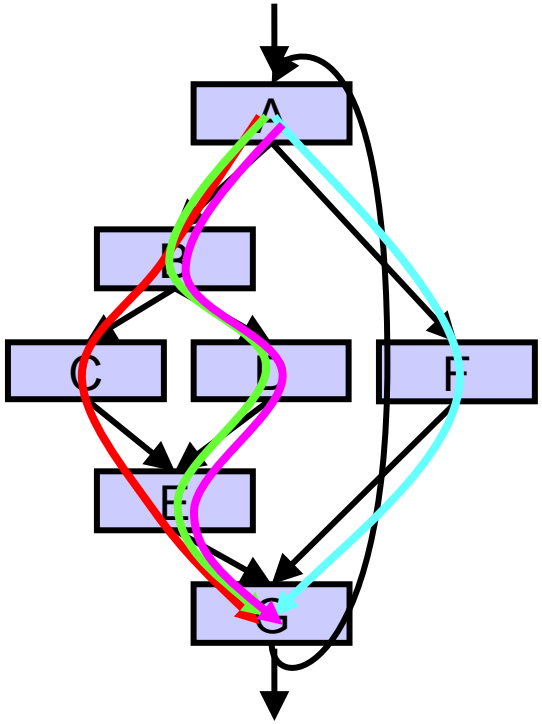
- Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Does not have to be lock step
  - Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
    - SW does not need to know vector length
    - Enables memory and branch latency tolerance
  - ISA is scalar → vector instructions formed dynamically
  - Essentially, it is SPMD programming model implemented on SIMD hardware

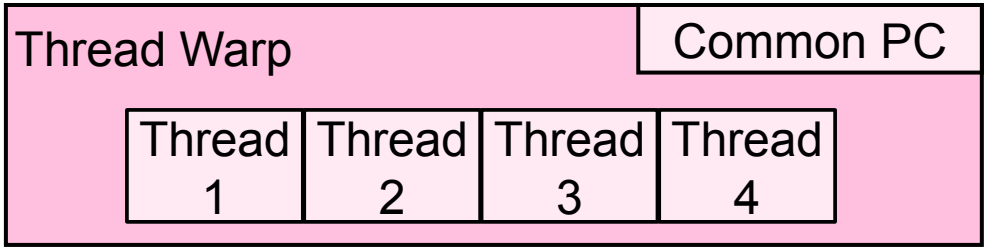
SPMD

- 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 can 1) execute a different control-flow path, 2) work on different data, at run-time
  - Many scientific applications programmed this way and run on MIMD computers (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD computer

Branch Divergence Problem in Warp-based SIMD

- SPMD Execution on SIMD Hardware
  - NVIDIA calls this “Single Instruction, Multiple Thread” (“SIMT”) execution

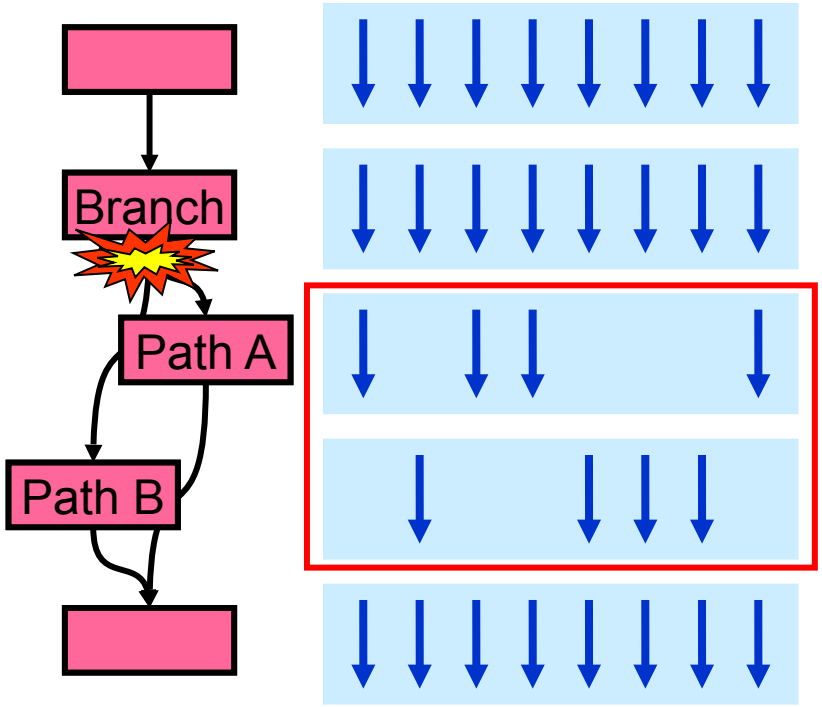




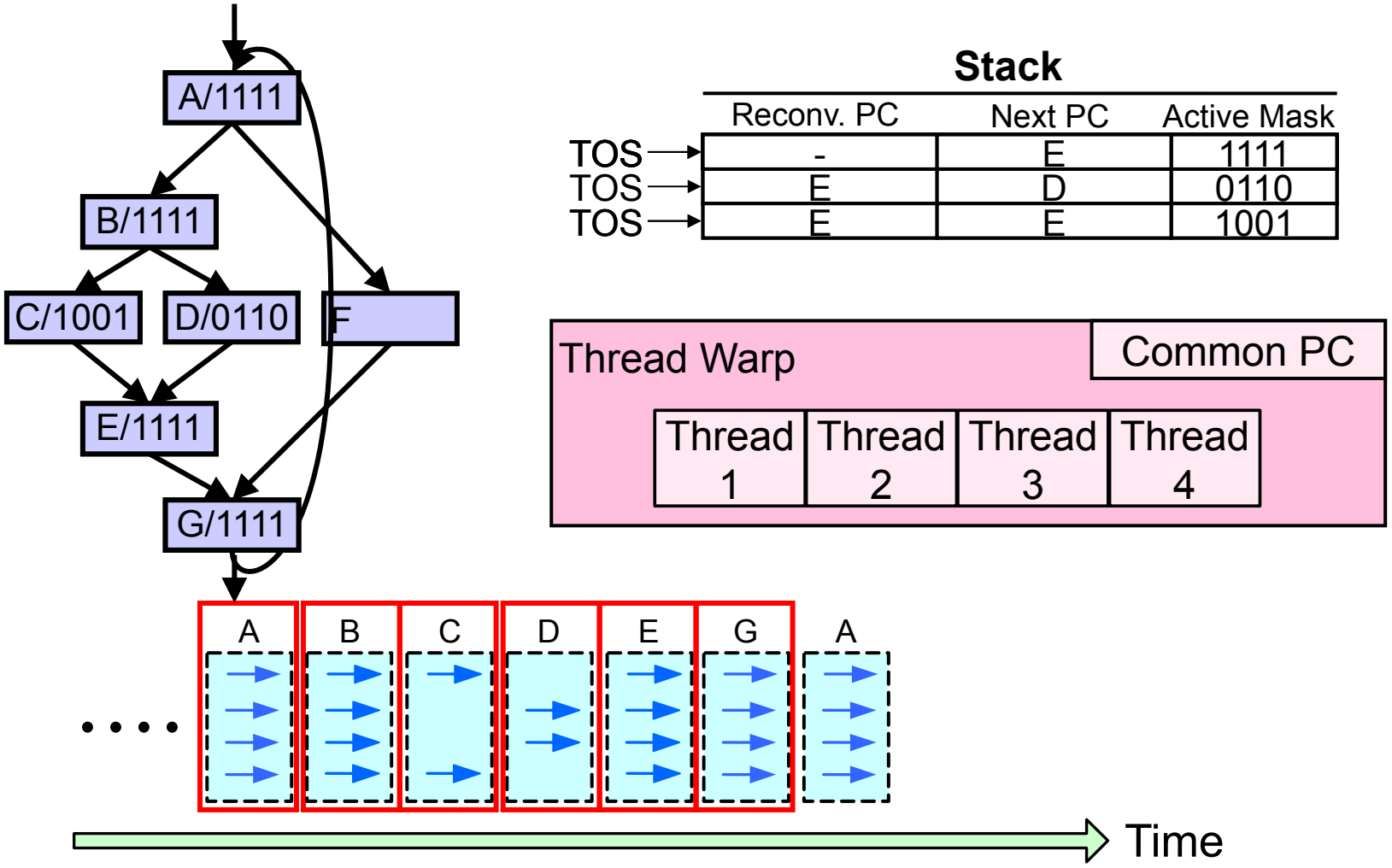
Slide credit: Tor Aamodt

Control Flow Problem in GPUs/SIMD

- GPU uses SIMD pipeline to save area on control logic.
  - Group scalar threads into warps
- Branch divergence occurs when threads inside warps branch to different execution paths.

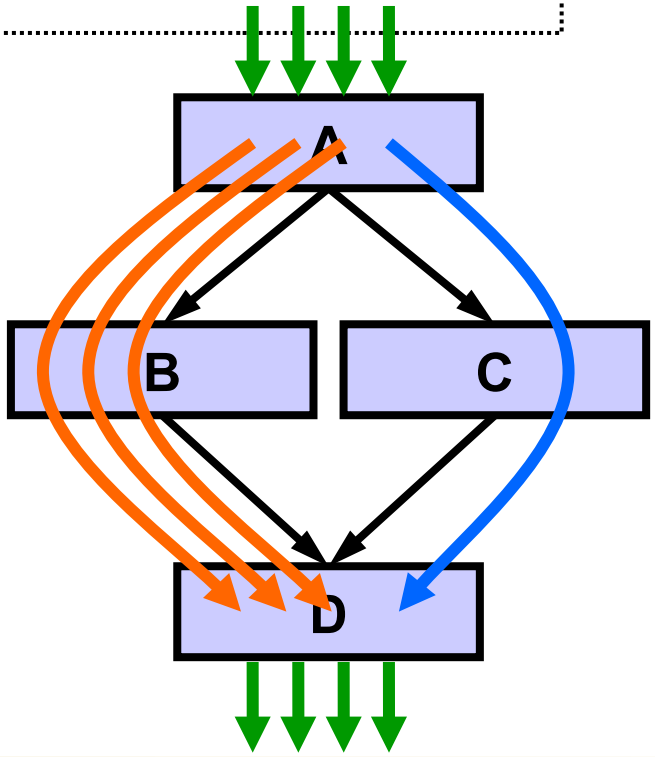


Branch Divergence Handling (I)



Branch Divergence Handling (II)

```
A;
if (some condition) {
   B;
} else {
   C;
}
D;
```

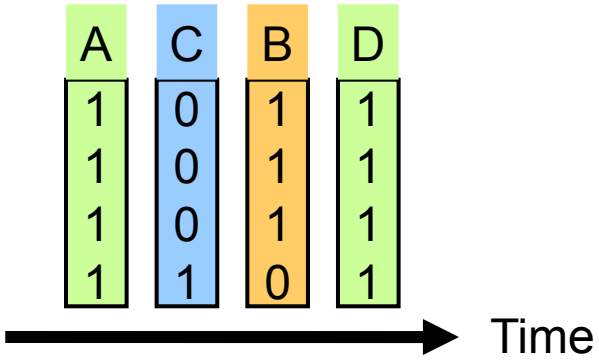


One per warp

Control Flow Stack

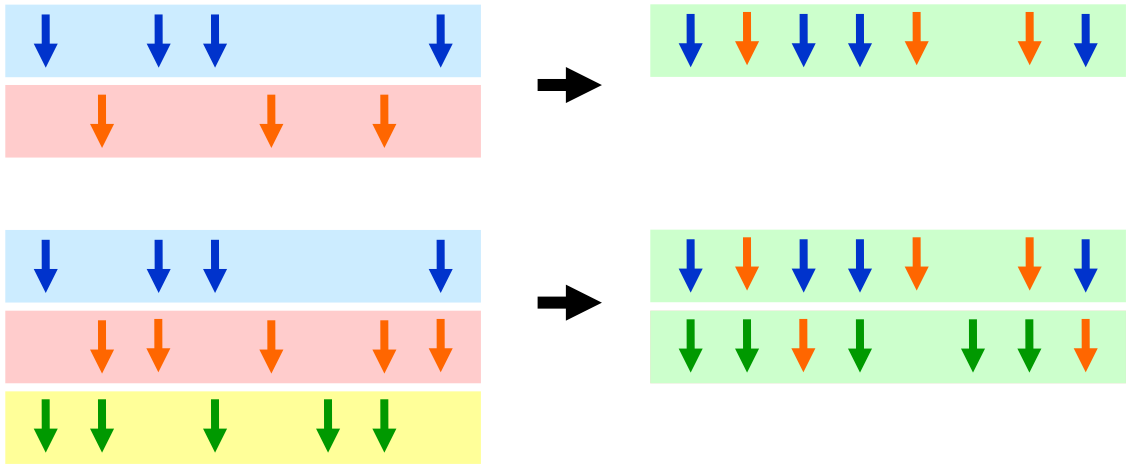
	Next PC	Recv PC	Amask
TOS →	D	--	1111
	B	D	1110
	D	D	0001

Execution Sequence



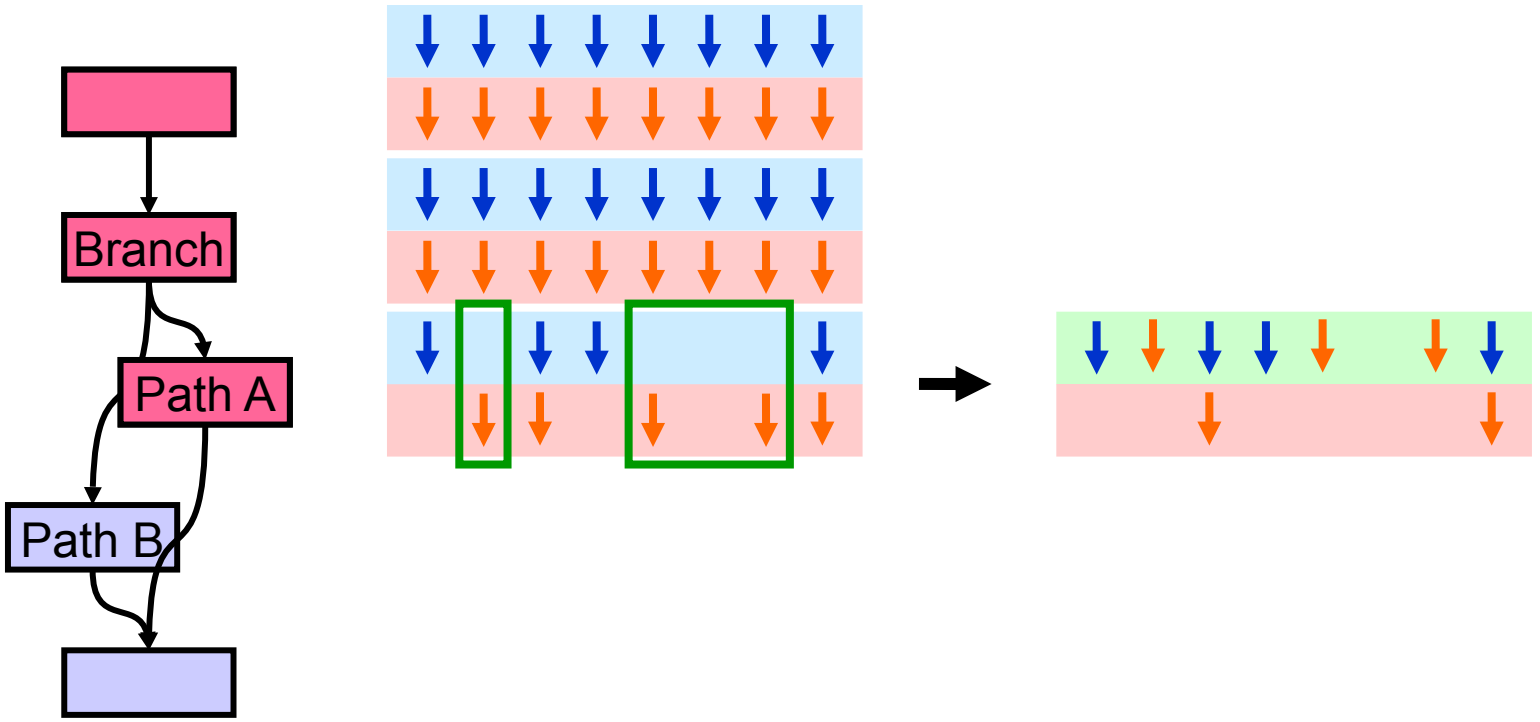
Dynamic Warp Formation

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)
- Form new warp at divergence
  - Enough threads branching to each path to create full new warps



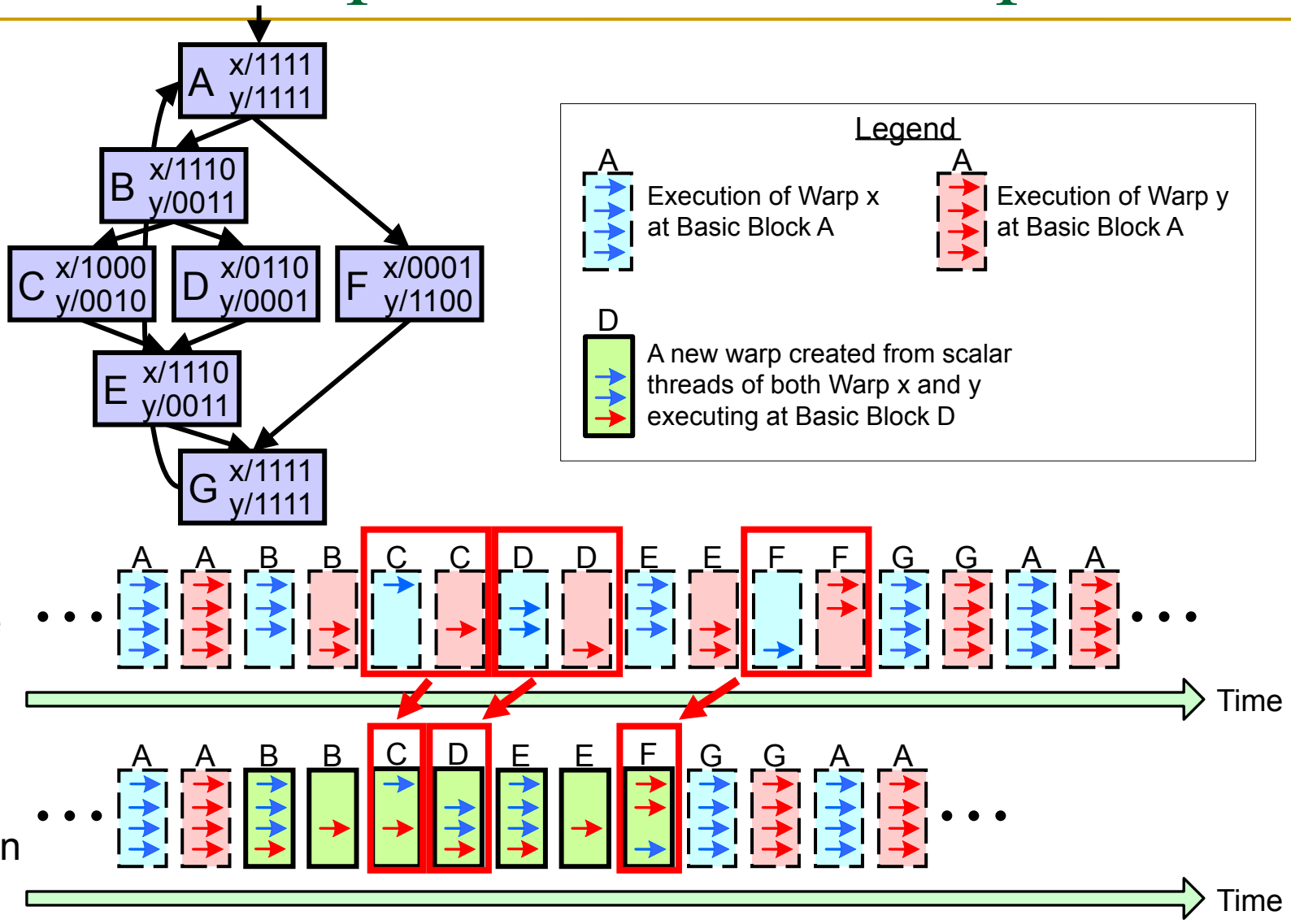
Dynamic Warp Formation/Merging

- 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



What About Memory Divergence?

- Modern GPUs have caches
- Ideally: Want all threads in the warp to hit (without conflicting with each other)
- Problem: One thread in a warp can stall the entire warp if it misses in the cache.

- Need techniques to
  - Tolerate memory divergence
  - Integrate solutions to branch and memory divergence