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

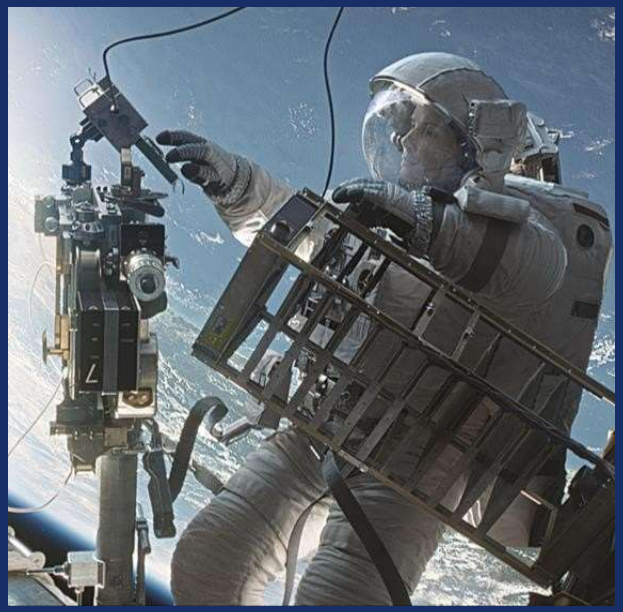
Xulong Tang

Slides adapted from Ashutosh Pattnaik (PSU)
Onur Mutlu (ETH), Sharan Chetlur (NVIDIA),
David Patterson, John L. Hennessy

PennState

INTRODUCING GPUs

I know what you're thinking…



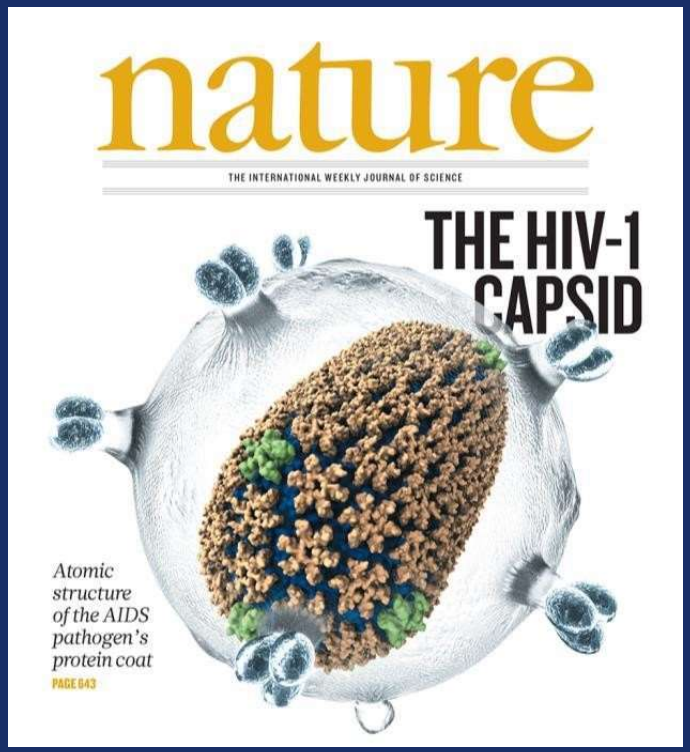
GAMING – SPECIAL FX

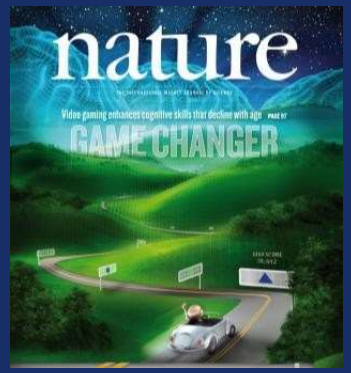
GPUs - ACCELERATING SCIENCE

Researchers using Tesla GPUs are solving the world's great scientific and technical challenges.

Using a supercomputer powered by 3,000 Tesla processors, University of Illinois scientists achieved a breakthrough in HIV research. Another research team from Baylor, Rice, MIT, Harvard and the Broad Institute used GPUs to map how the human genome folds within the nucleus of a cell.

These and other advances in science have been highlighted in top journals .





Agenda

- Overview of Data Level Parallel Architectures
  - SIMD Processing
  - SIMD support in modern ISAs
  - Graphics Processing Units

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 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

Instruction Stream

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

ARRAY PROCESSOR

Same op @ same time

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

Different ops @ same space

Time

Space

VECTOR PROCESSOR

Different ops @ time

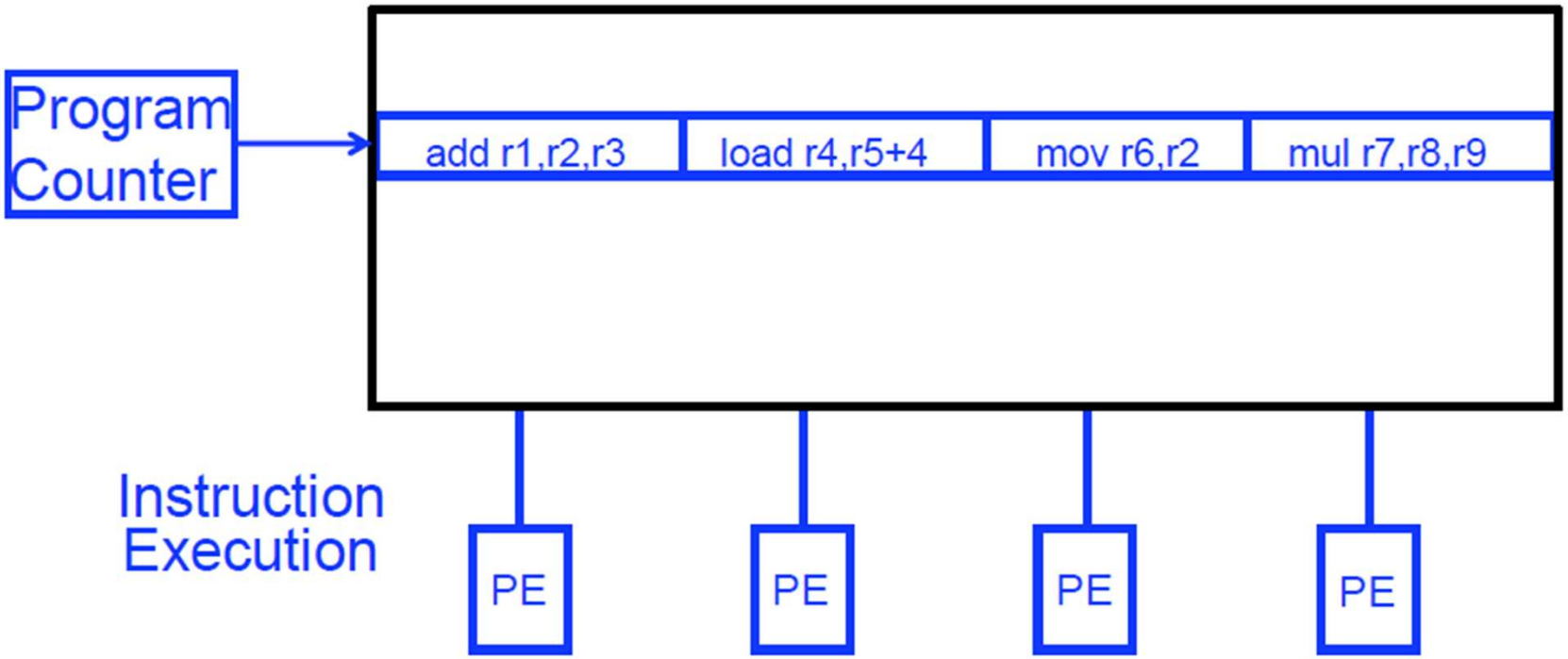
LD0			
LD1	AD0		
LD2	AD1	MU0	
LD3	AD2	MU1	ST0
	AD3	MU2	ST1
		MU3	ST2
			ST3

Same op @ space

Space

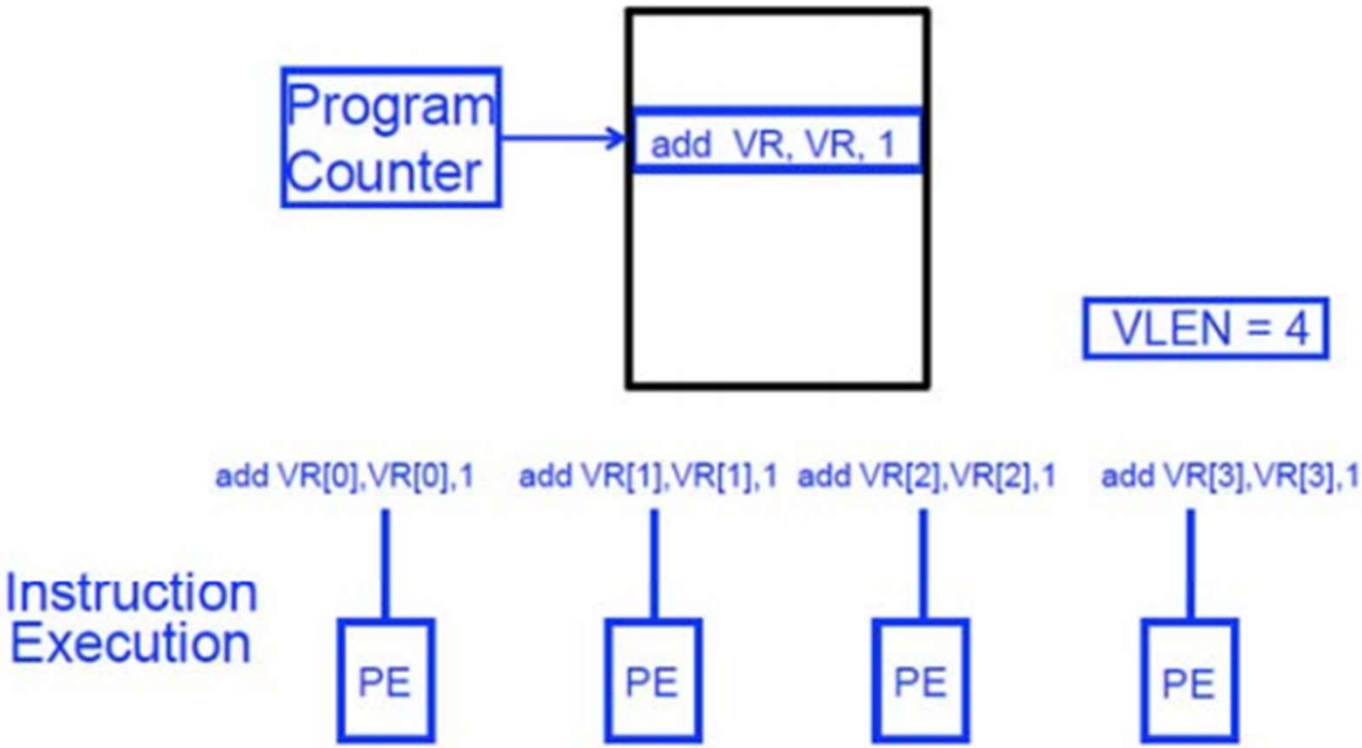
SIMD Array Processing vs. VLIW

- VLIW (Very Large Instruction Word)



SIMD Array Processing vs. VLIW

- VLIW (Very Large Instruction Word)



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

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!

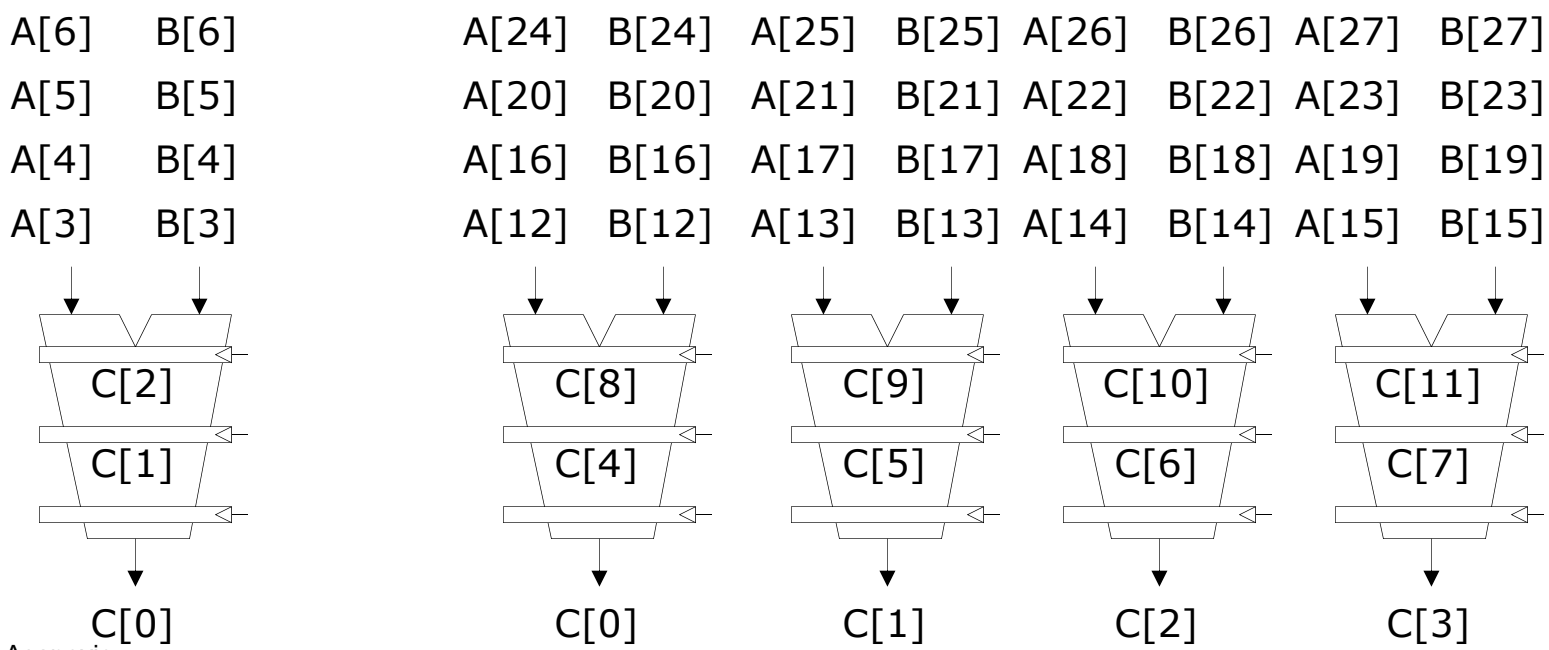
- Many existing ISAs include (vector-like) SIMD operations
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD

Vector Instruction Execution

ADDV C,A,B

Execution using one pipelined functional unit

Execution using four pipelined functional units

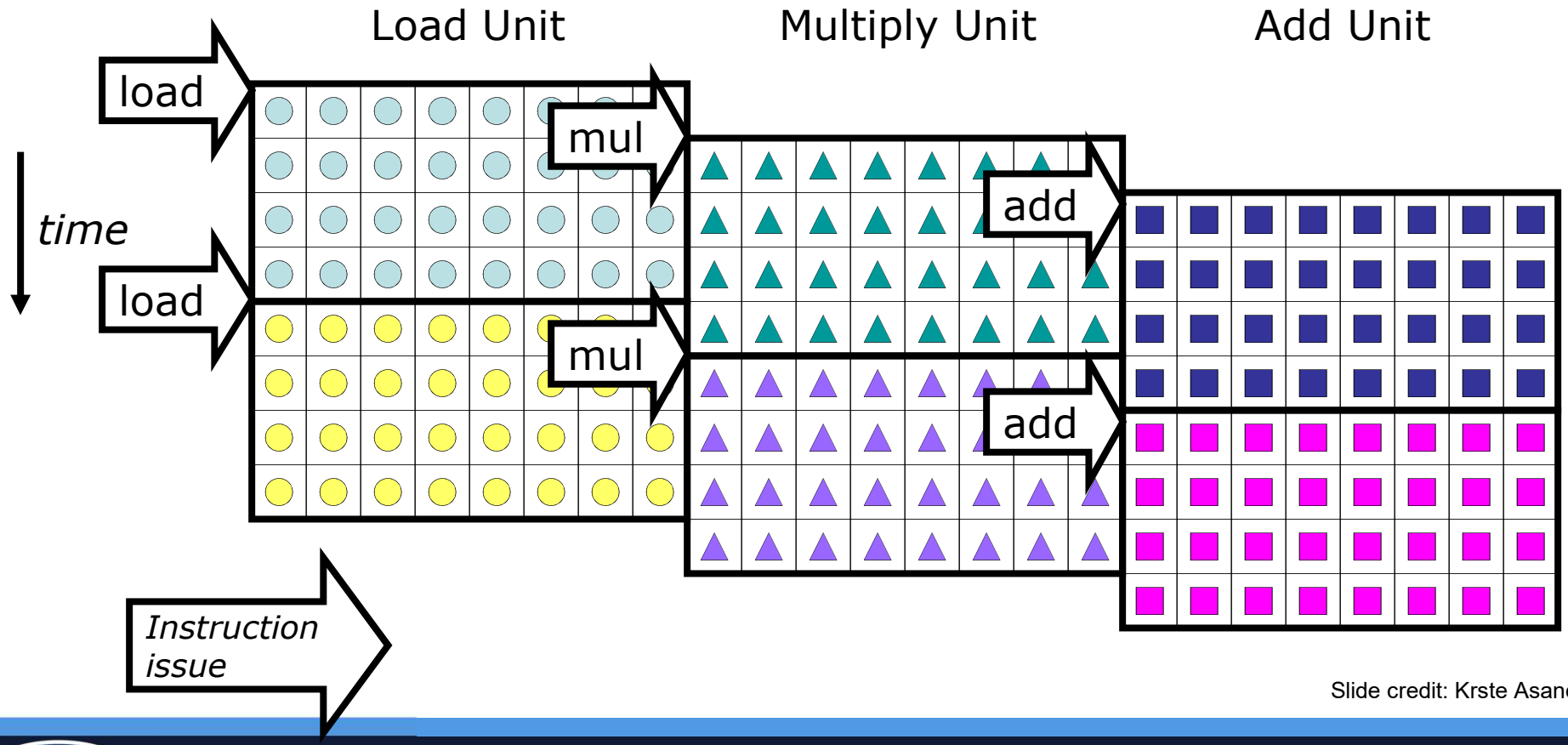


Slide credit: Krste Asanovic

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



Slide credit: Krste Asanovic

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

Finally!

Graphics Processing Units:

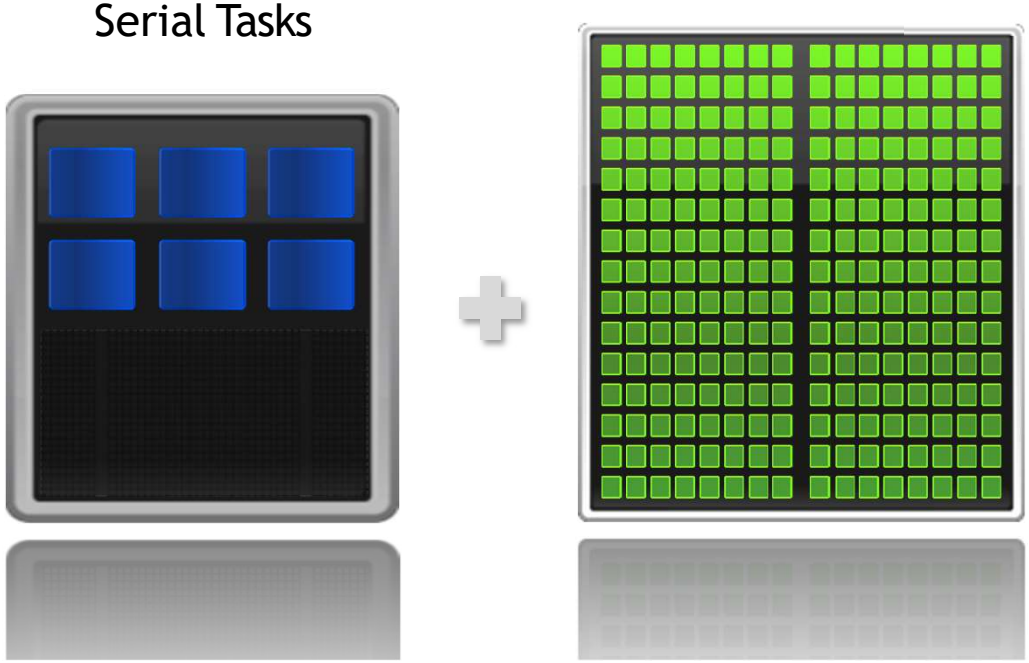
SIMD not Exposed to Programmer (SIMT)

GPU ACCELERATED COMPUTING

10X PERFORMANCE 5X ENERGY EFFICIENCY

CPU
Optimized for
Serial Tasks

GPU Accelerator
Optimized for
Parallel Tasks

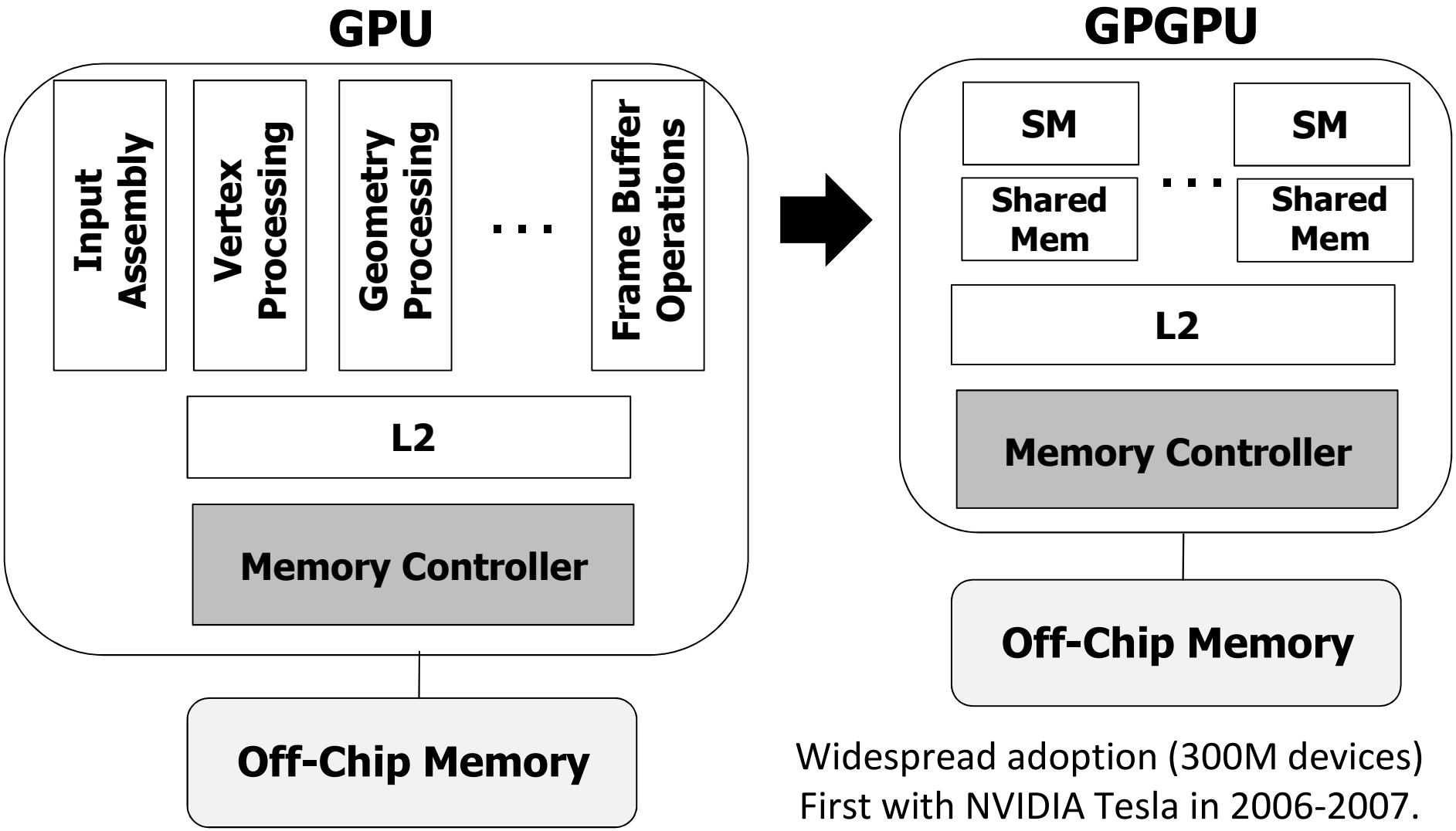


Graphical Processing Units

- Given the hardware invested to do graphics well, how can we supplement it to improve performance of a wider range of applications?

- Basic idea:
  - Heterogeneous execution model
    - CPU is the *host*, GPU is the *device*
  - Develop a C-like programming language for GPU
  - Unify all forms of GPU parallelism as *CUDA thread*
  - Programming model is "Single Instruction Multiple Thread"

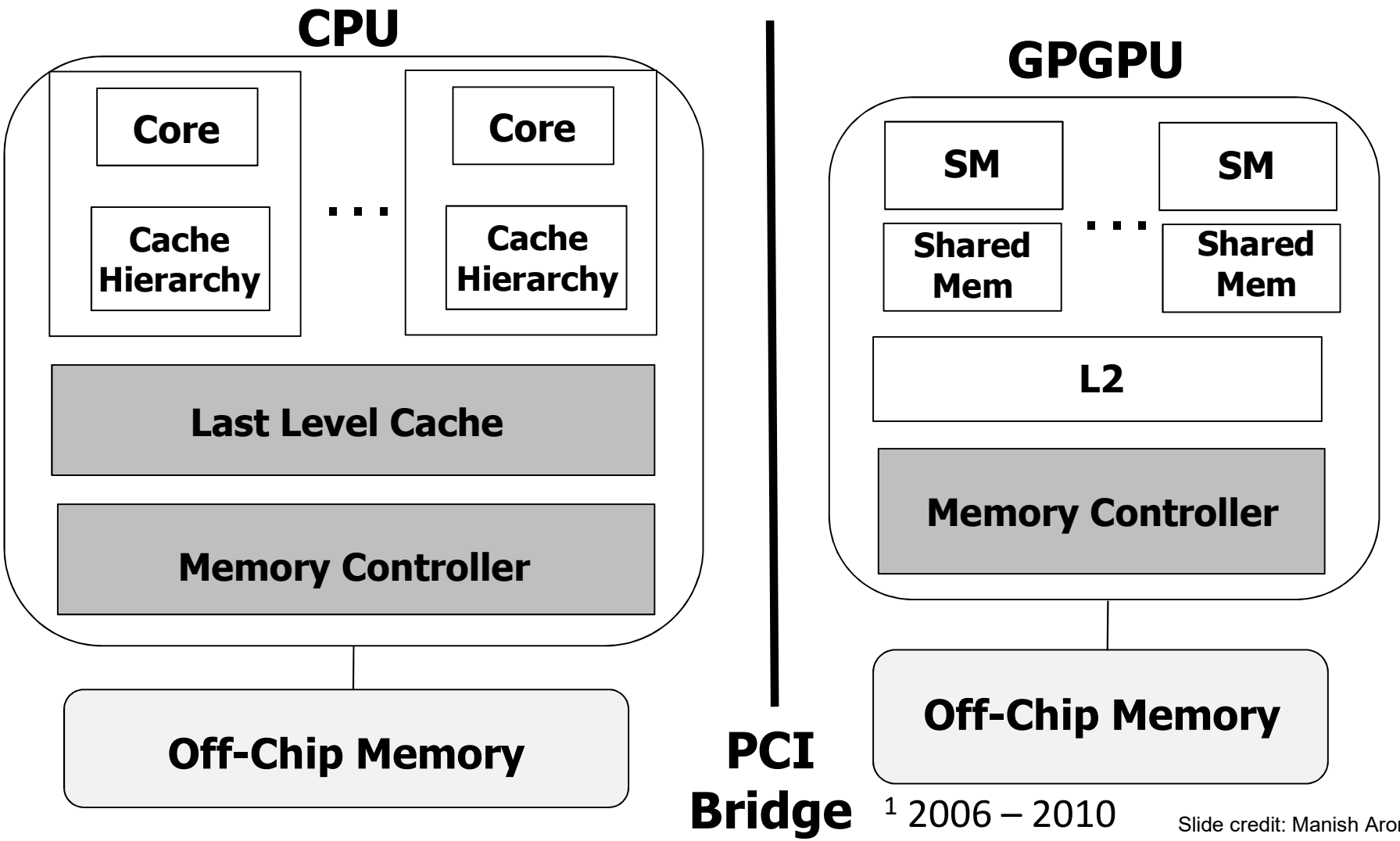
From GPU to GPGPU



Widespread adoption (300M devices)
First with NVIDIA Tesla in 2006-2007.

Slide credit: Manish Arora

Consumer Hardware[1]



[1] 2006 – 2010

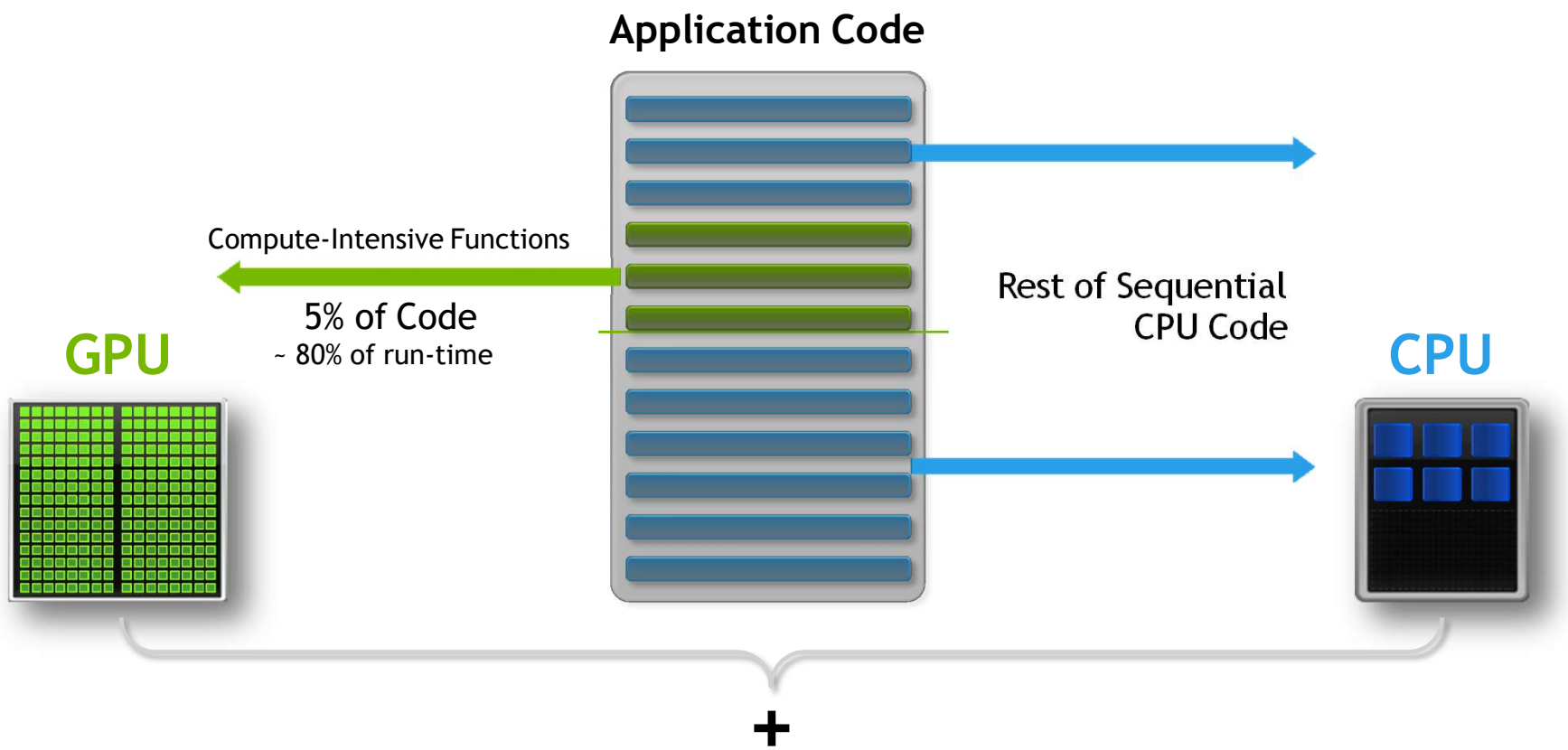
Slide credit: Manish Arora

NVIDIA GPU Architecture

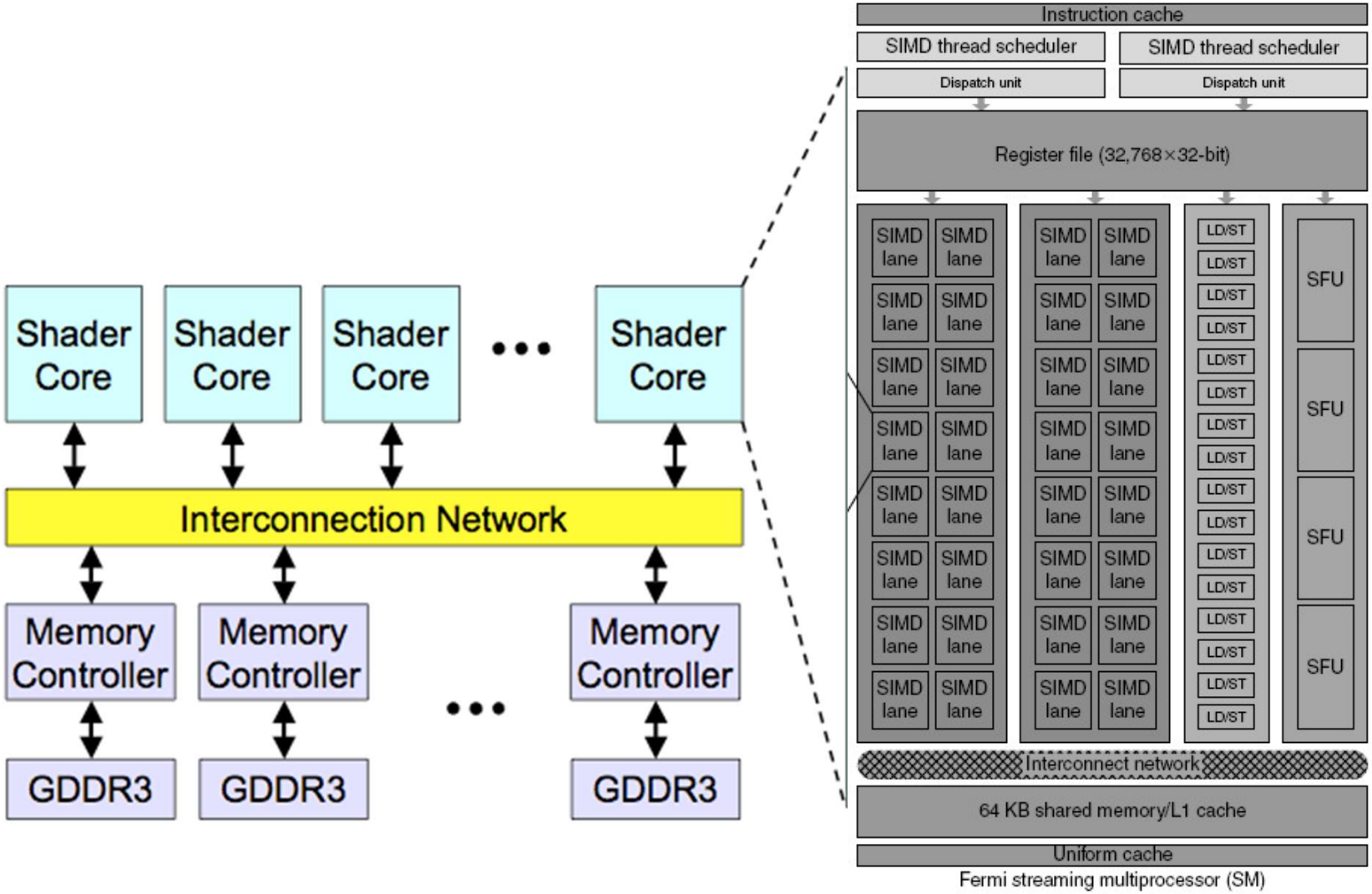
- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files

- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

HOW GPU ACCELERATION WORKS



High-Level View of a GPU



Introduction to CUDA Programming

- **Declspecs**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function launch**

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

See https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

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

Threads and Blocks

- A thread is associated with each data element
- Threads are organized into blocks
- Blocks are organized into a grid

- GPU hardware handles thread management, not applications or OS

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)

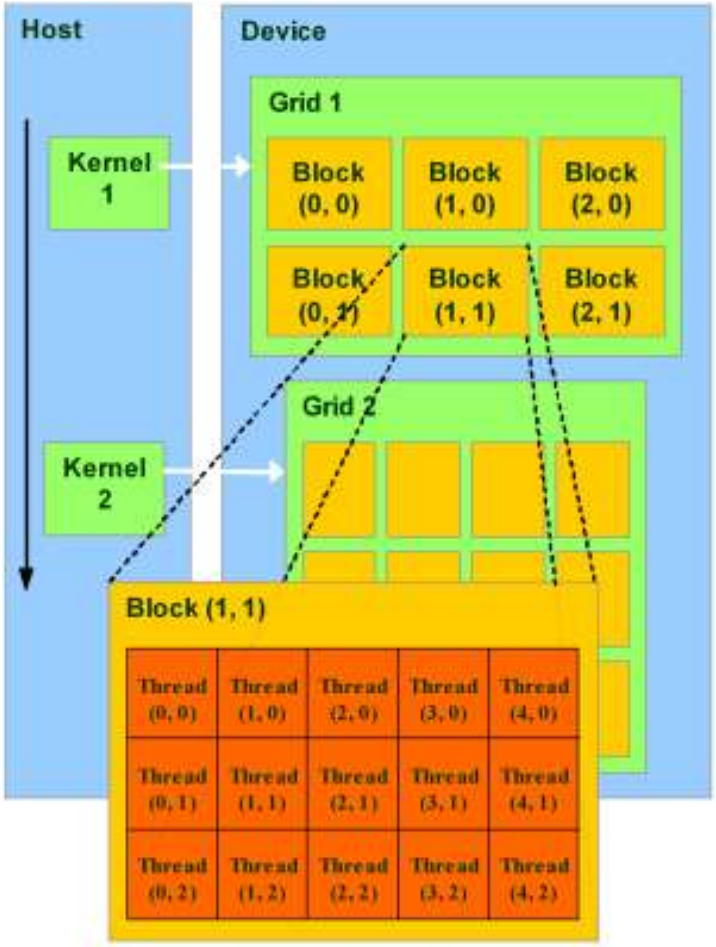
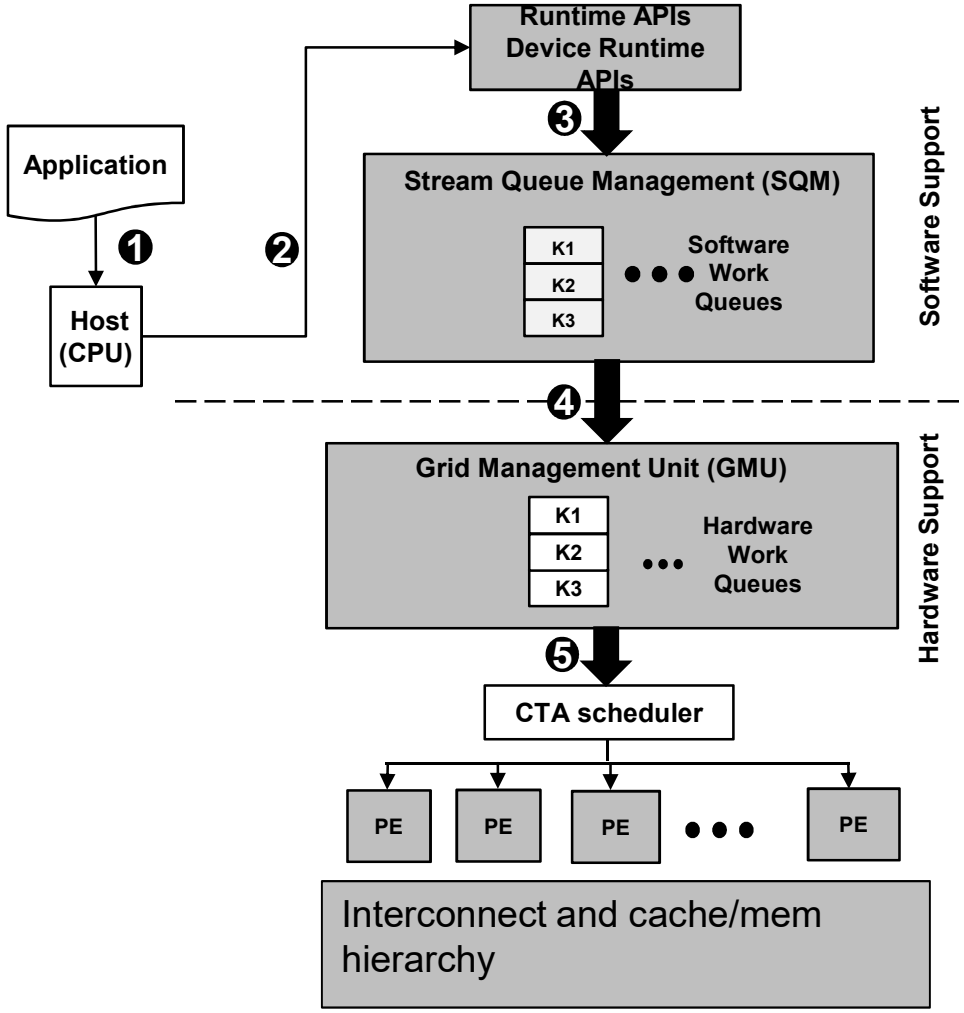


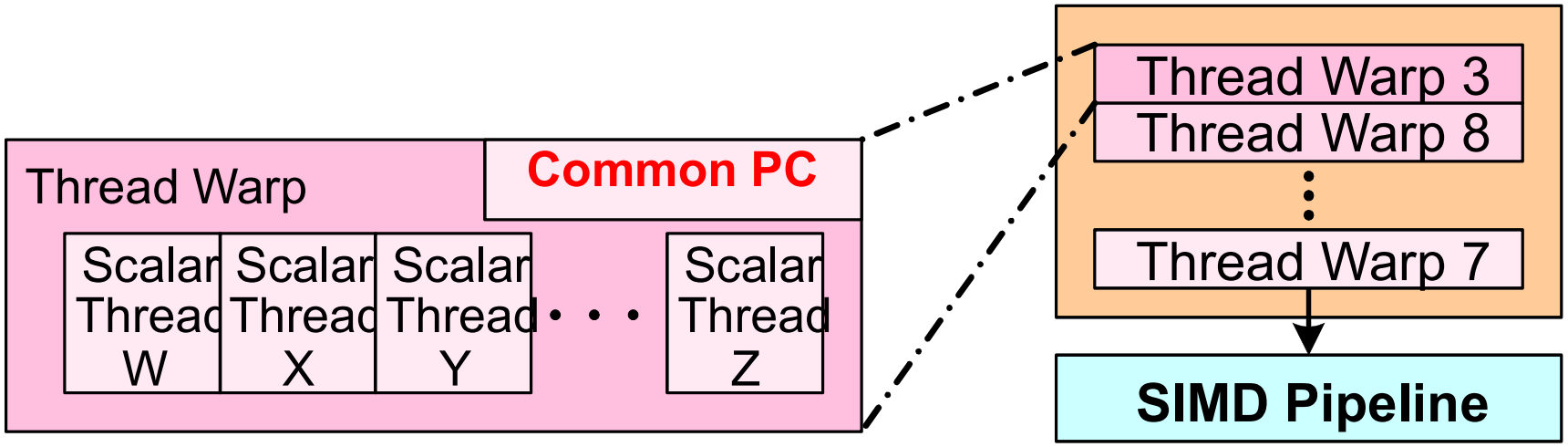
Image credit: http://ixbtlabs.com/

CUDA Execution



Concept of "Thread Warps" and SIMT

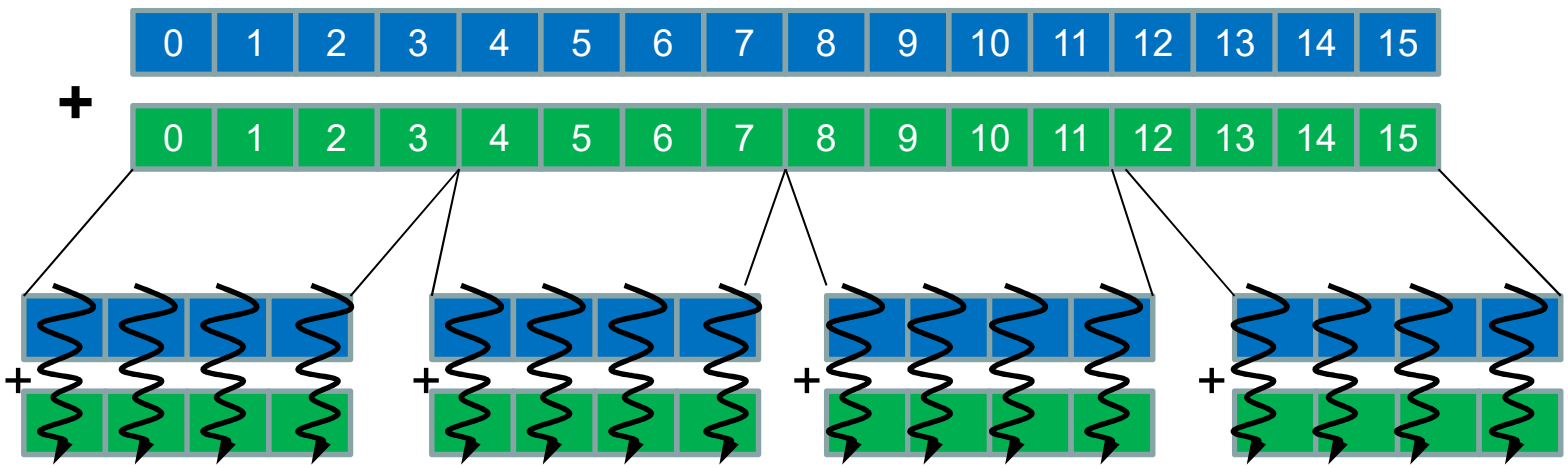
- Warp: A set of threads that execute the same instruction (on different data elements) → SIMT (Nvidia-speak)
- All threads run the same kernel
- Warp: The threads that run lengthwise in a woven fabric …



SIMT Memory Access

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

Let's assume N=16, blockDim=4 → 4 blocks

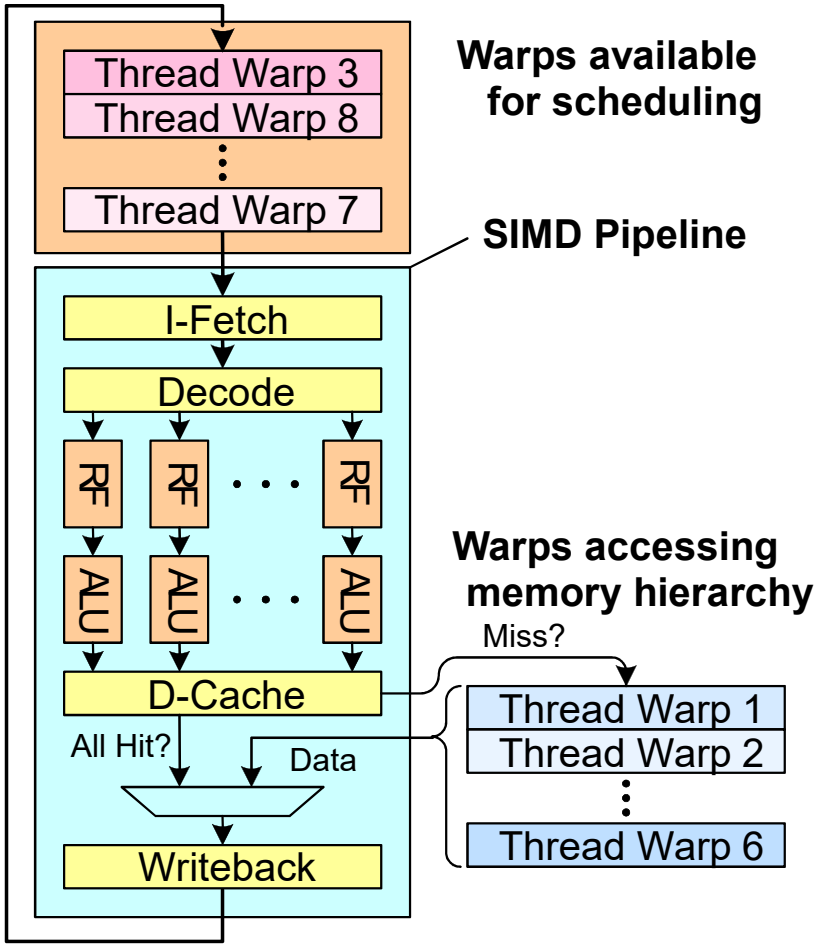


Slide credit: Hyesoon Kim

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



Slide credit: Tor Aamodt

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

Thank you!

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

Slides adapted from Onur Mutlu (ETH), Sharan Chetlur (NVIDIA), David Patterson, John L. Hennessy

PennState