Data-Level Parallelism

(Computer Architecture: Chapter 4)

Yanyan Shen

Department of Computer Science and Engineering

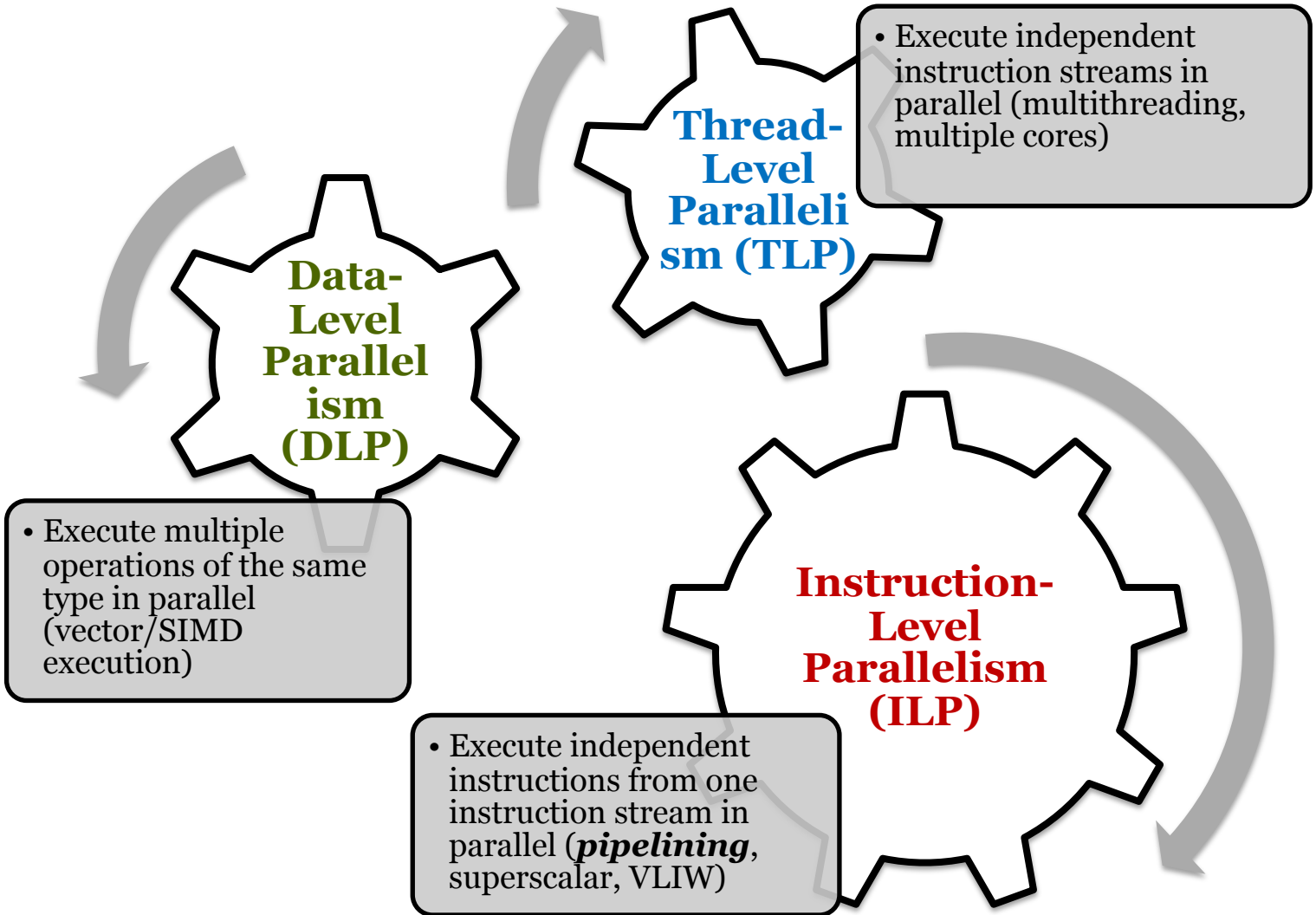
Shanghai Jiao Tong University

Outline

- 4.1 Introduction
- 4.2 Vector Architecture
- 4.3 SIMD Instruction Set Extensions
- 4.4 GPU

Flynn's Classification (1966)

Approaches to Exploiting Parallelism



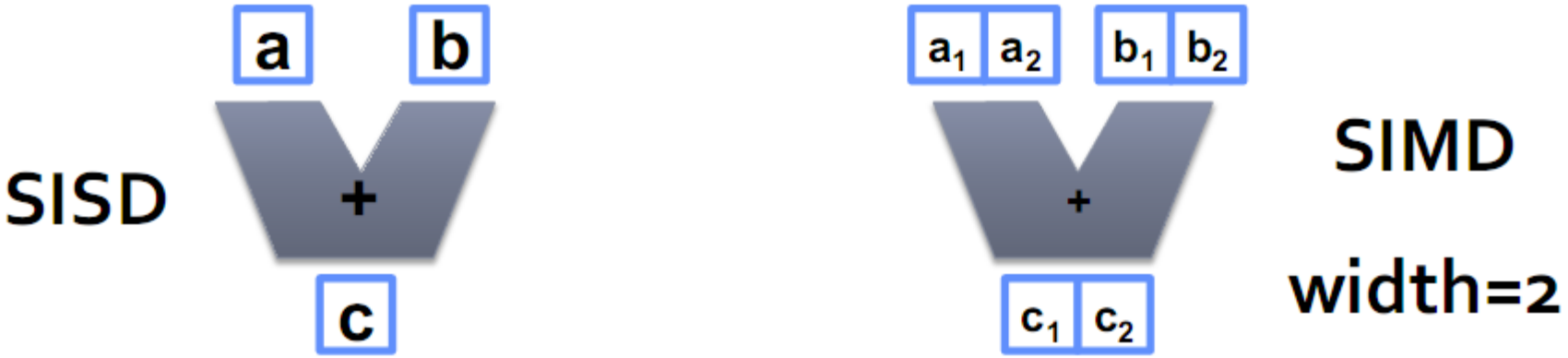
Resurgence of DLP

- Convergence of *application demands* and *technology constraints* drives architecture choice
- New applications, *such as graphics, machine vision, speech recognition, machine learning, etc.* all require large numerical computations that are often **trivially data parallel**
- SIMD-based architectures (vector-SIMD, subword-SIMD, GPUs) are most efficient ways to execute these algorithms

Introduction

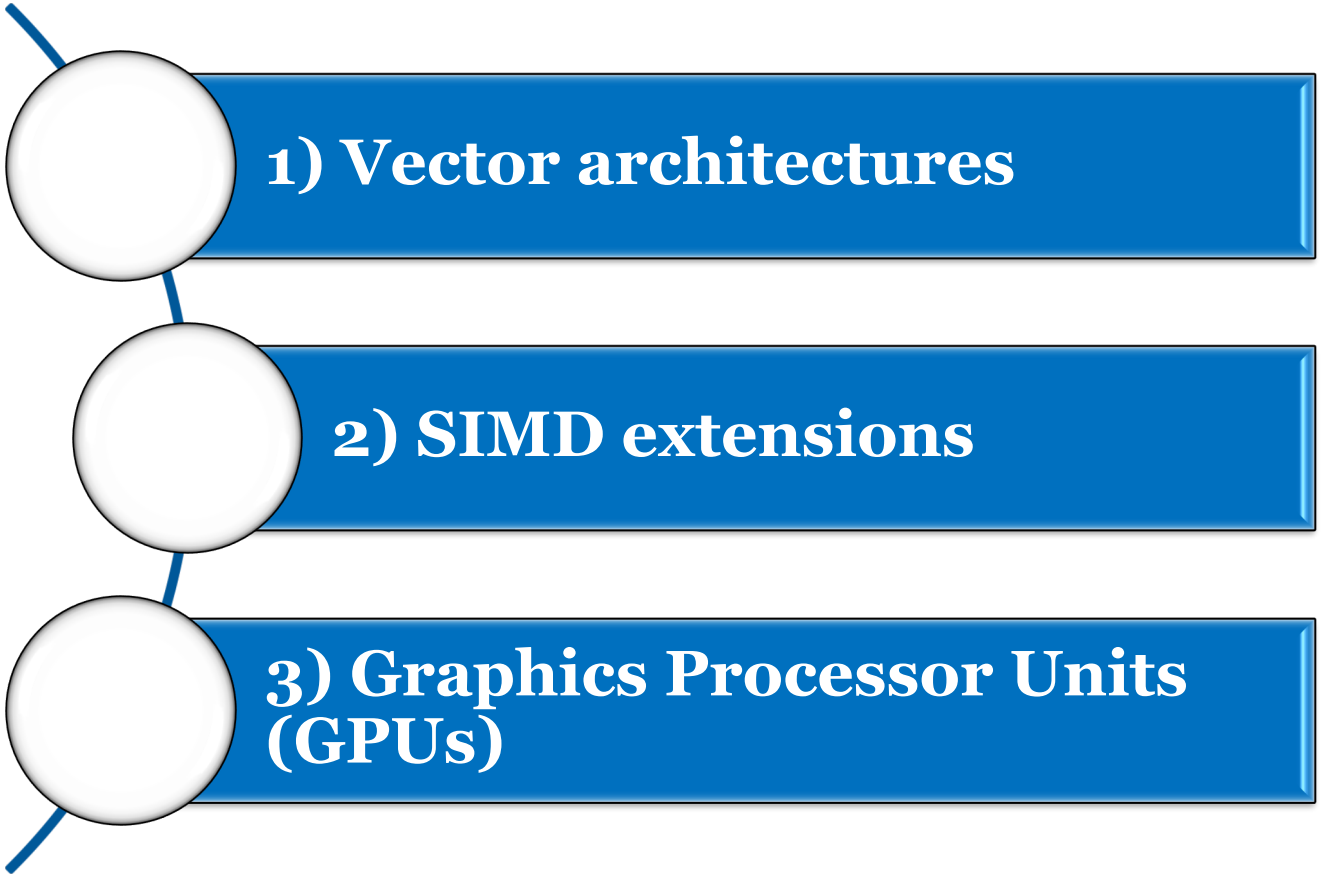
- SIMD architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processors
- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction for data operations
  - Makes SIMD attractive for personal mobile devices
- **Advantage** compared to MIMD: SIMD allows programmer to continue to think sequentially

SIMD



- *Single Instruction Multiple Data* architectures make use of data parallelism
- We care about SIMD because of area and power efficiency concerns
  - – Amortize control overhead over SIMD width
- However, parallelism exposed to programmer & compiler

SIMD Parallelism



Outline

- 4.1 Introduction
- 4.2 Vector Architecture
- 4.3 SIMD Instruction Set Extensions
- 4.4 GPU

Vector Architectures

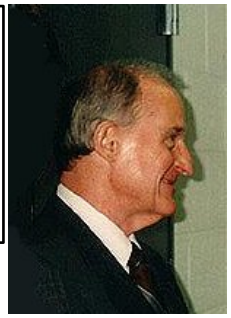
- **Basic idea:**
  - Read sets of data elements into "**vector registers**"
  - Operate on those registers
  - Disperse the results back into memory

- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth

Vector Supercomputers

Seymour Roger Cray

Father of supercomputers



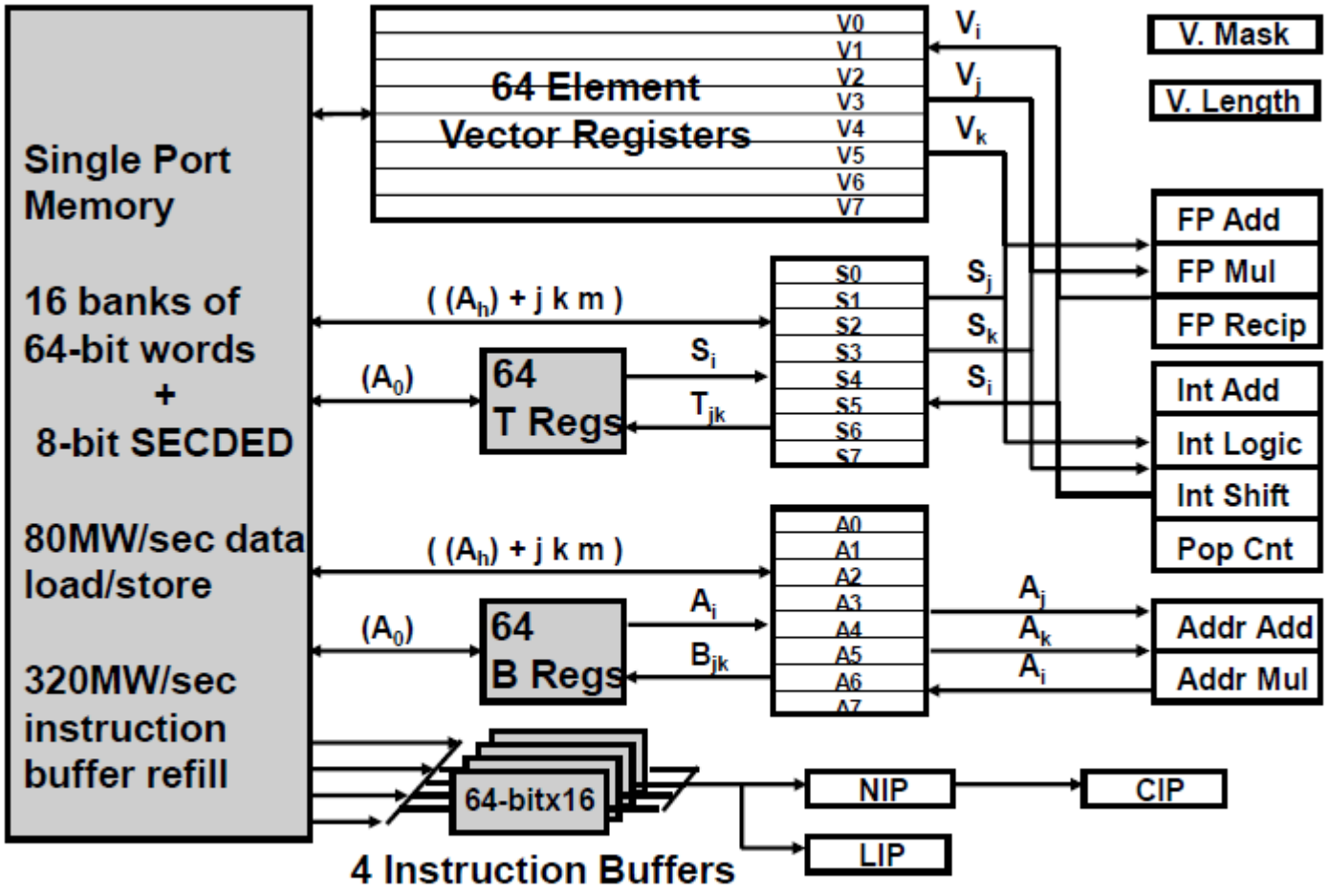
Epitomized by Cray-1, 1976:

Scalar Unit + Vector Extensions

- Load/Store Architecture
- Vector Registers
- Vector Instructions
- Hardwired Control
- Highly Pipelined Functional Units
- Interleaved Memory System
- No Data Caches
- No Virtual Memory

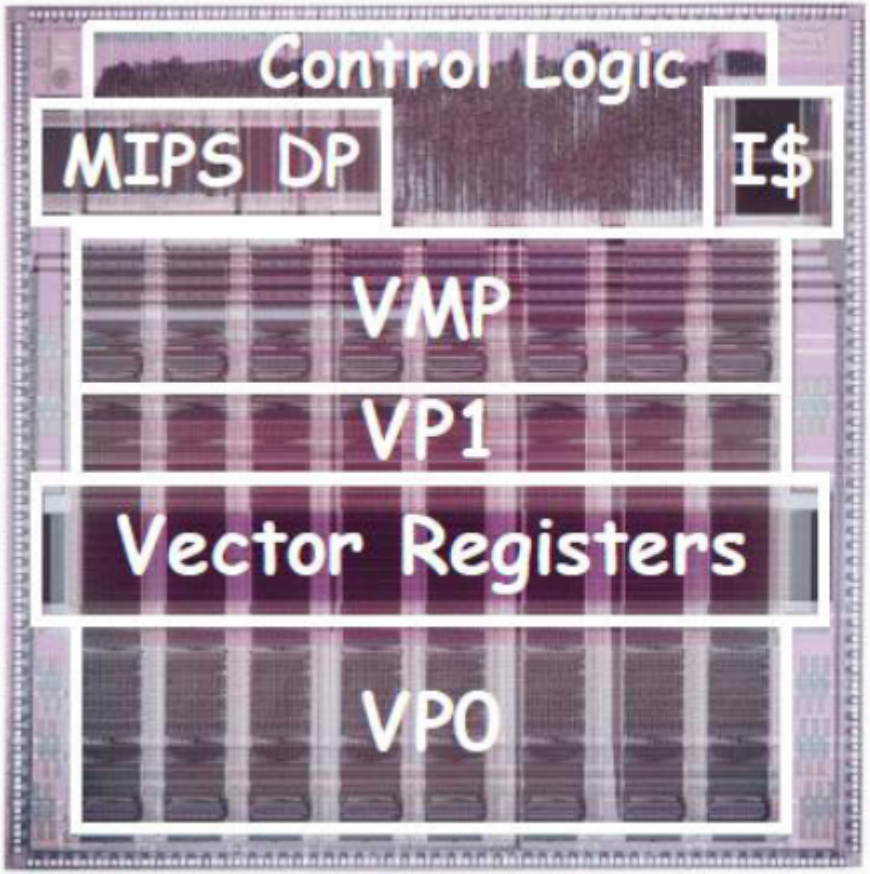
The initial model weighed 5.5 tons including the Freon refrigeration system

Cray-1 (1976)



memory bank cycle 50 ns *processor cycle* 12.5 ns (80MHz)

To (Torrent) Vector Microprocessor (1995)



Switched to HP CMOS 26G process late in design

- used 1.0μm rules in 0.8μm process
- only used 2 out of 3 metal layers

16.75x16.75mm^2

730,701 transistors

4W typical @ 5V, 40MHz

12W maximum

Performance:

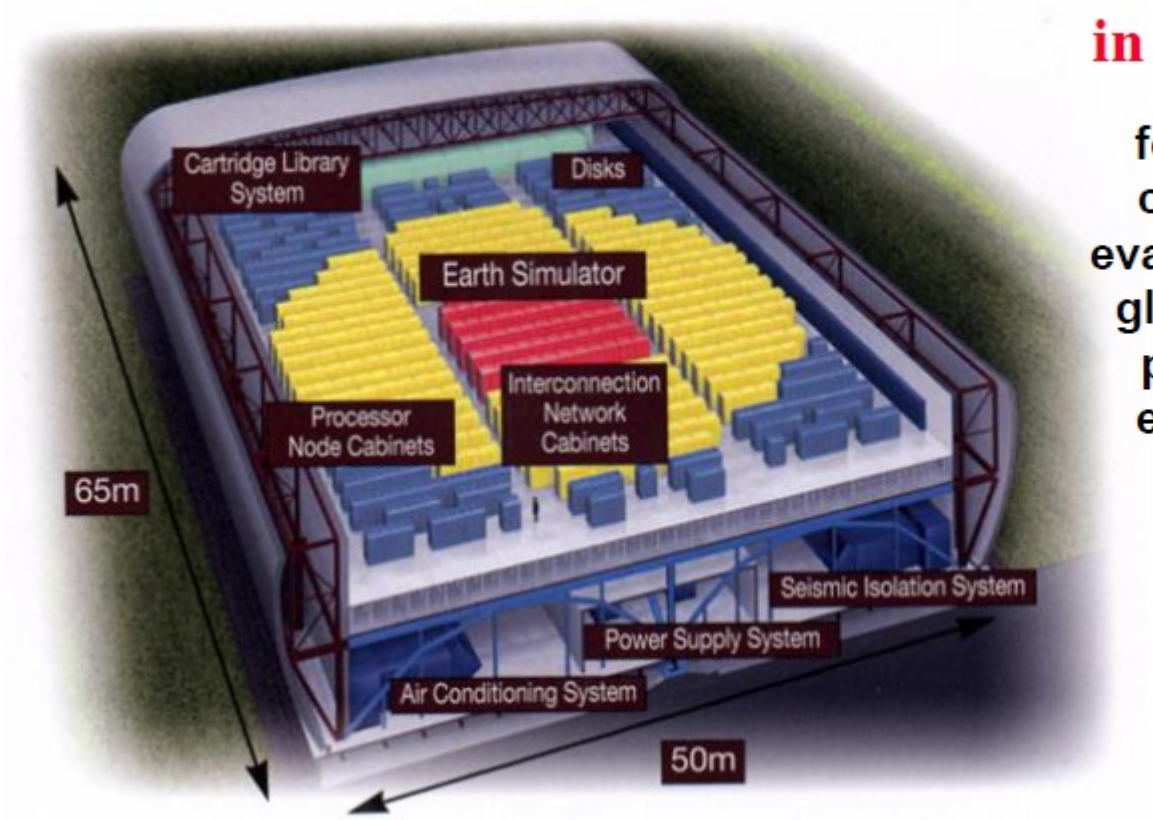
320MMAC/s

640MB/s

Vector Supercomputer: NEC SX-6 (2003)

- CMOS Technology
  - Each 500 MHz CPU fits on single chip
  - SDRAM main memory (up to 64GB)
- Scalar unit in each CPU
  - 4-way superscalar with out-of-order and speculative execution
  - 64KB I-cache and 64KB data cache
- Vector unit in each CPU
  - 8 foreground VRegs + 64 background VRegs (256x64-bit elements/VReg)
  - 1 multiply unit, 1 divide unit, 1 add/shift unit, 1 logical unit, 1 mask unit
  - 8 lanes (8 GFLOPS peak, 16 FLOPS/cycle)
  - 1 load & store unit (32x8 byte accesses/cycle)
  - 32 GB/s memory bandwidth per processor
- SMP (Symmetric Multi-Processor) structure
  - 8 CPUs connected to memory through crossbar
  - 256 GB/s shared memory bandwidth (4096 interleaved banks)

Earth System Simulator (NEC, ~2002)



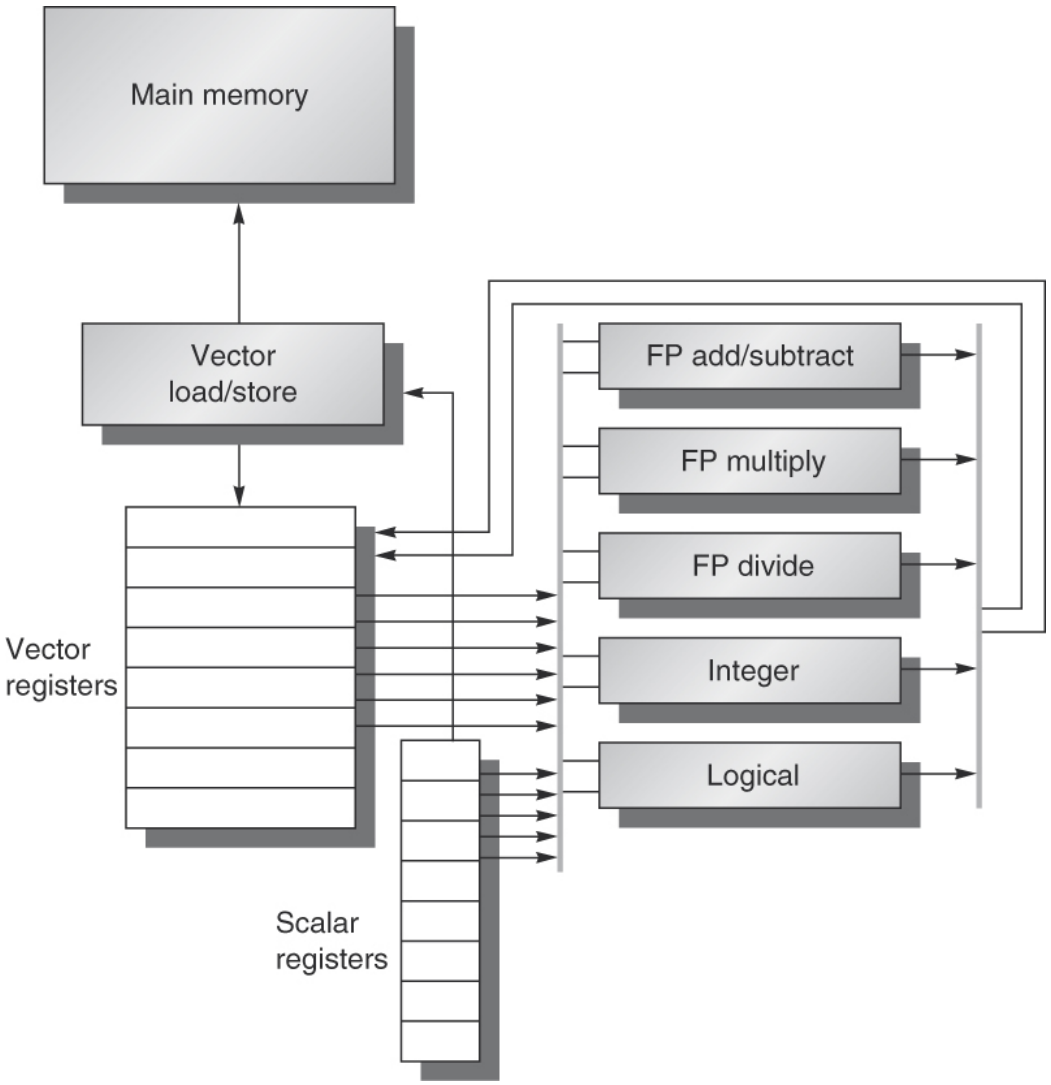
Fastest computer in world, 2002-04

for running global climate models to evaluate the effects of global warming and problems in solid earth geophysics.

640 compute nodes, each with 8 vector processors; 10 TB memory

Performance: 40 TFLOPS

Basic structure of VMIPS



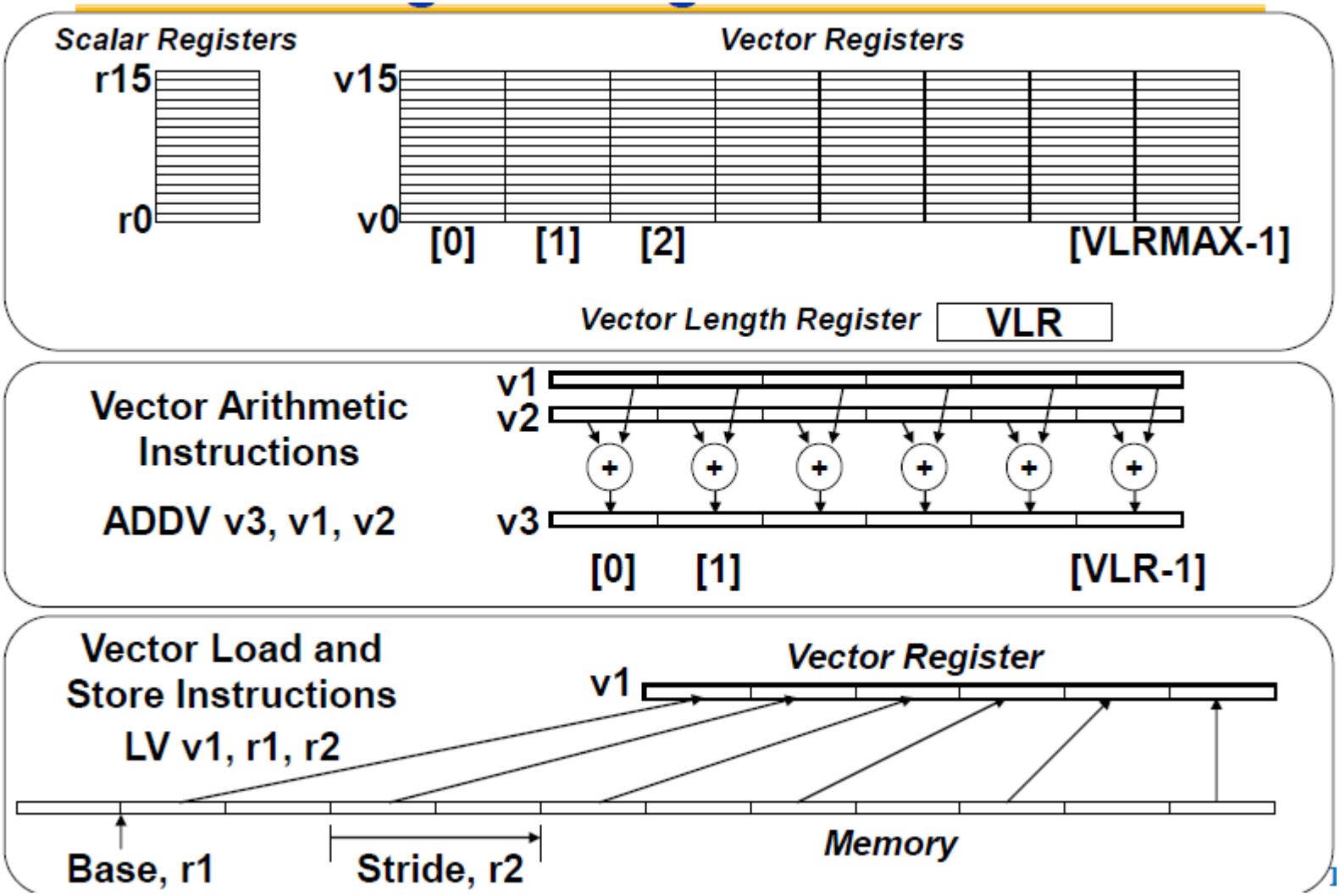
- This processor has a scalar architecture just like MIPS.
- There are also eight 64-element vector registers, and vector functional units.
- The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations.
- A set of crossbar switches (thick gray lines) connects these ports to the inputs and outputs of the vector functional units.

Architecture: VMIPS (running example)

Loosely based on Cray-1

- Vector registers
  - Each register holds a *64-element, 64 bits/element vector*
  - Register file has *16 read ports and 8 write ports*
- Vector functional units
  - Fully pipelined
  - Data and control hazards are detected
- Vector load-store unit
  - Fully pipelined
  - One word per clock cycle after initial latency
- Scalar registers
  - 32 general-purpose registers
  - 32 floating-point registers

Vector Programming Model



Vector Instructions

Instruction	Operands	Function
ADDVV.D	V1,V2,V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1,V2,F0	Add F0 to each element of V2, then put each result in V1.
SUBVV.D	V1,V2,V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1,V2,F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1,F0,V2	Subtract elements of V2 from F0, then put each result in V1.
MULVV.D	V1,V2,V3	Multiply elements of V2 and V3, then put each result in V1.
MULVS.D	V1,V2,F0	Multiply each element of V2 by F0, then put each result in V1.
DIVVV.D	V1,V2,V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1,V2,F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1,F0,V2	Divide F0 by elements of V2, then put each result in V1.
LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1,V1	Store vector register V1 into memory starting at address R1.
LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2 (i.e., R1 + i × R2).
SVWS	(R1,R2),V1	Store V1 to address at R1 with stride in R2 (i.e., R1 + i × R2).
LVI	V1,(R1+V2)	Load V1 with vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
SVI	(R1+V2),V1	Store V1 to vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
CVI	V1,R1	Create an index vector by storing the values 0, 1 × R1, 2 × R1,...,63 × R1 into V1.
S--VV.D S--VS.D	V1,V2 V1,F0	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.
POP	R1,VM	Count the 1s in vector-mask register VM and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1 MFC1	VLR,R1 R1,VLR	Move contents of R1 to vector-length register VL. Move the contents of vector-length register VL to R1.
MVTM MVFM*	VM,F0 F0,VM	Move contents of F0 to vector-mask register VM. Move contents of vector-mask register VM to F0.

ADDVV.D: add two vectors

ADDVS.D: add vector to a scalar

LV/SV: vector load and vector store from address

Vector Instruction Set Advantages

- **Compact**
  - one short instruction encodes N operations
- **Expressive**, tells hardware that these N operations:
  - are independent
  - use the same functional unit
  - access disjoint registers (elements)
  - access registers in the same pattern as previous instructions
  - access a contiguous block of memory (unit-stride load/store)
  - access memory in a known pattern (strided load/store)
- **Scalable**
  - can run same object code on more parallel pipelines or lanes
  - i.e., the code is machine independent

Example: DAXPY

$$Y = a \times X + Y$$

DAXPY:
Double Precision a*X plus Y

64 elements in total

How Vector Processor Work: An Example

MIPS code:

```
        L.D     F0,a            ;load scalar a
        DADDIU  R4,Rx,#512      ;last address to load
Loop:   L.D     F2,0(Rx)        ;load X[i]
        MUL.D   F2,F2,F0        ;a × X[i]
        L.D     F4,0(Ry)        ;load Y[i]
        ADD.D   F4,F4,F2        ;a × X[i] + Y[i]
        S.D     F4,9(Ry)        ;store into Y[i]
        DADDIU  Rx,Rx,#8        ;increment index to X
        DADDIU  Ry,Ry,#8        ;increment index to Y
        DSUBU   R20,R4,Rx       ;compute bound
        BNEZ    R20,Loop        ;check if done
```

How Vector Processor Work: An Example

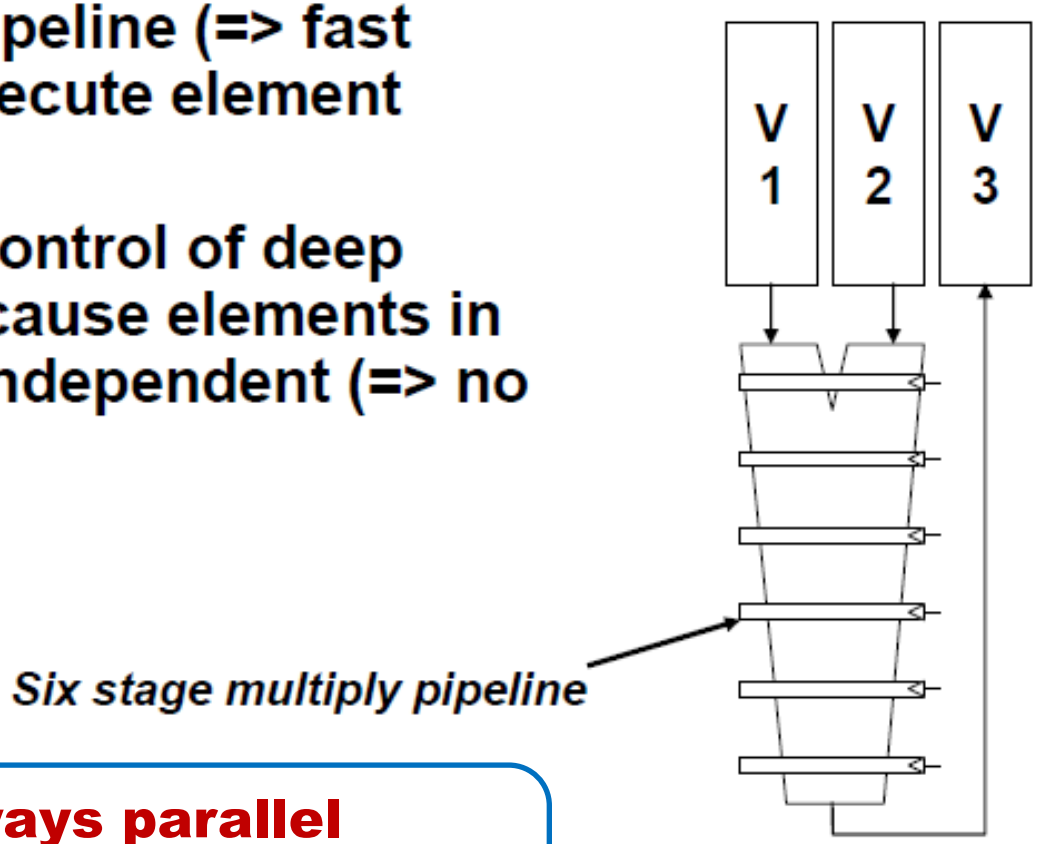
VMIPS code:

```
L.D       F0,a        ;load scalar a
LV        V1,Rx       ;load vector X
MULVS.D   V2,V1,F0    ;vector-scalar multiply
LV        V3,Ry       ;load vector Y
ADDVV.D   V4,V2,V3    ;add
SV        V4,Ry       ;store the result
```

600 instructions -> 6 instructions

Vector Arithmetic Execution

- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)



Note: not always parallel hardware (1 lane)

Deep pipeline also helps!

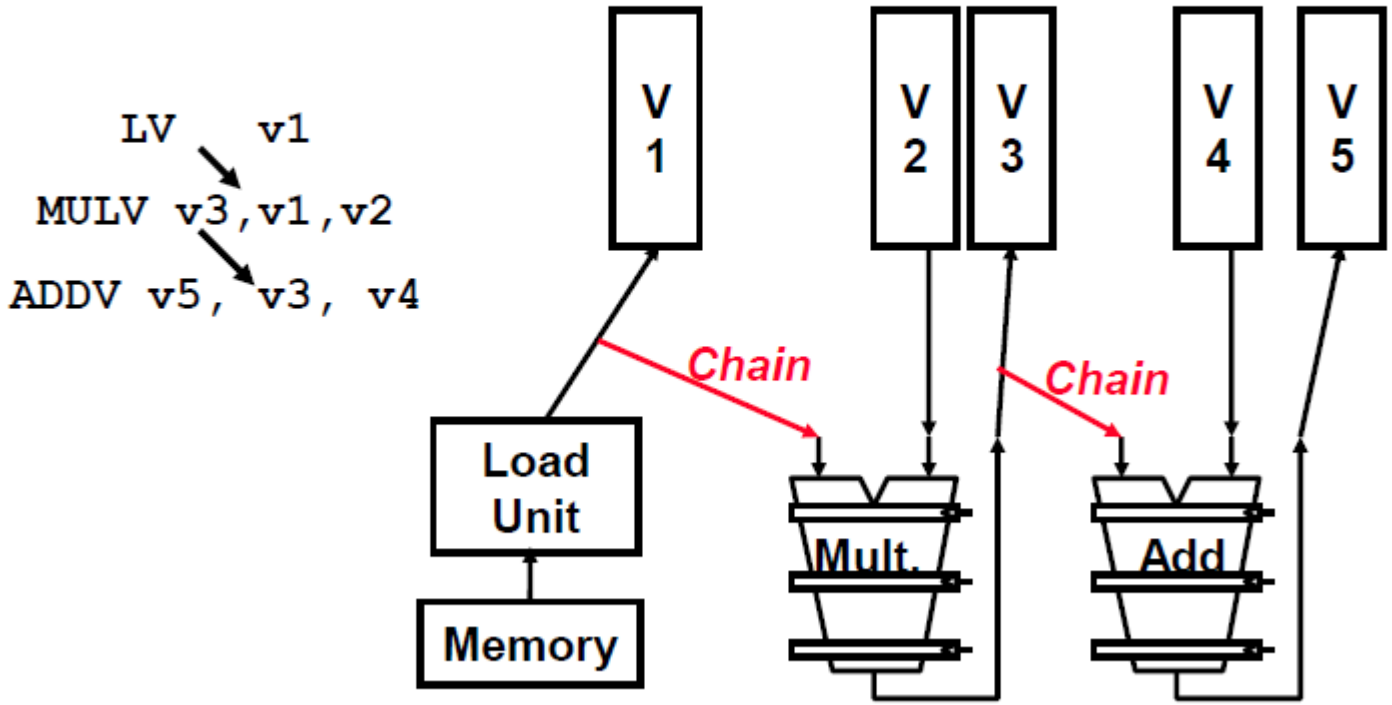
Vector Execution Time

- Execution time of a sequence of vector operations depends ***on three factors***:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies

- VMIPS functional units consume one element per clock cycle
  - Execution time is approximately the vector length

Vector Chaining

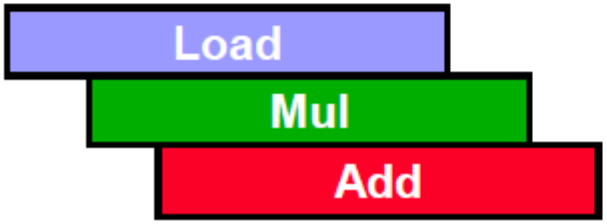
- Vector version of register bypassing
  - introduced with Cray-1



Vector Chaining Advantage

Without chaining, must wait for last element of result to be written before starting dependent instruction

With chaining, can start dependent instruction as soon as first result appears

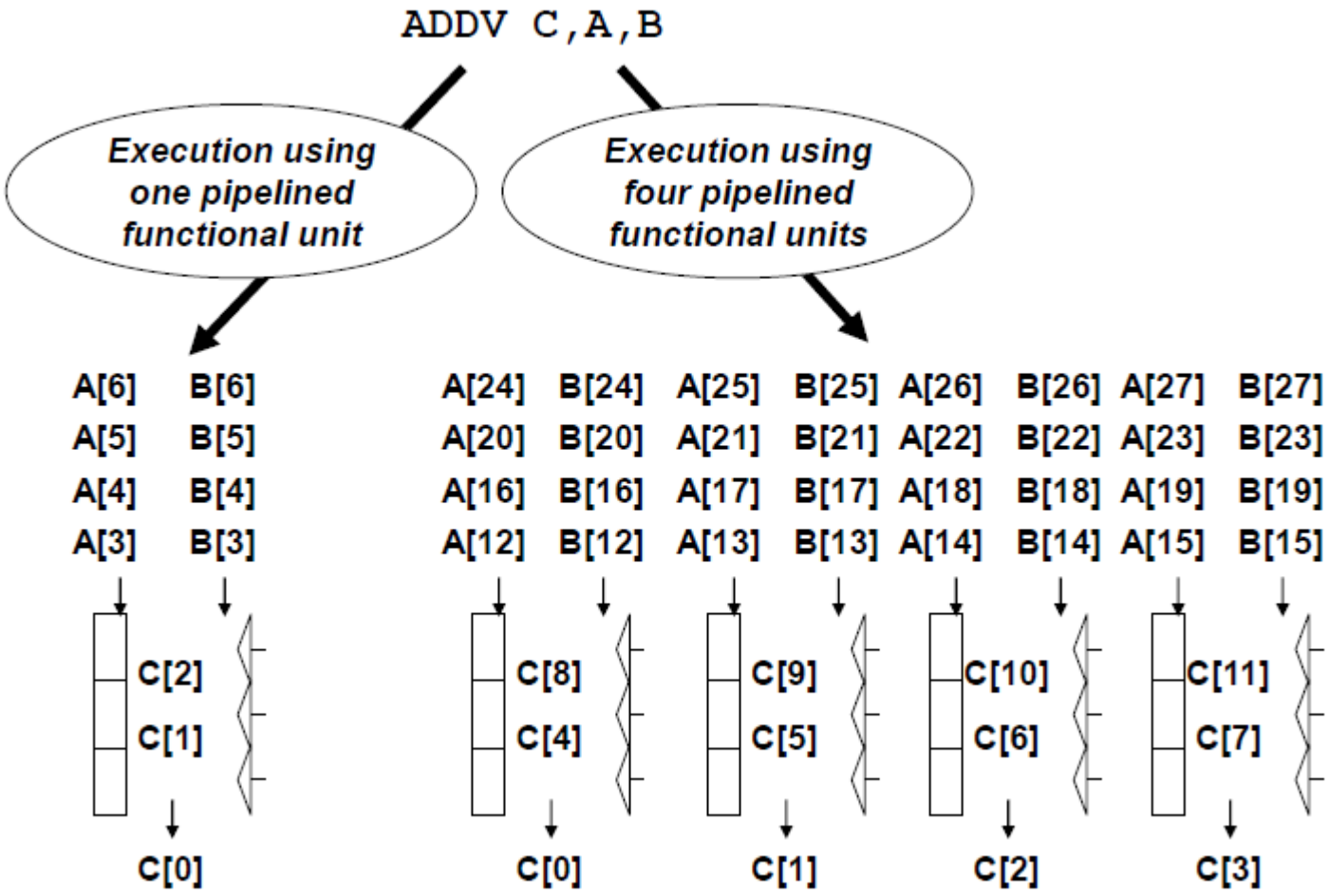


Optimizations

How can a vector processor execute a single vector faster than one element per clock cycle?

Multiple elements per clock cycle improve performance

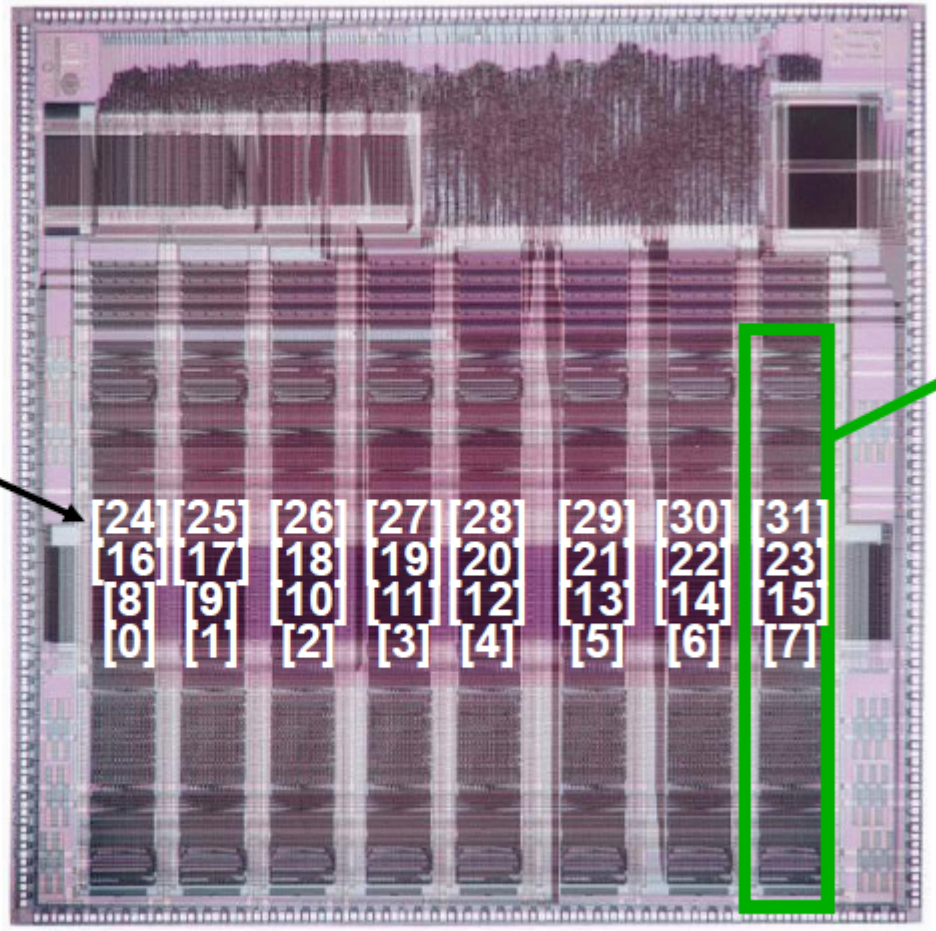
Vector Instruction Execution



To (Torrent) Vector Microprocessor (1995)

Idea: Add a vector coprocessor to a standard RISC scalar processor, all on one chip

Vector register elements striped over lanes



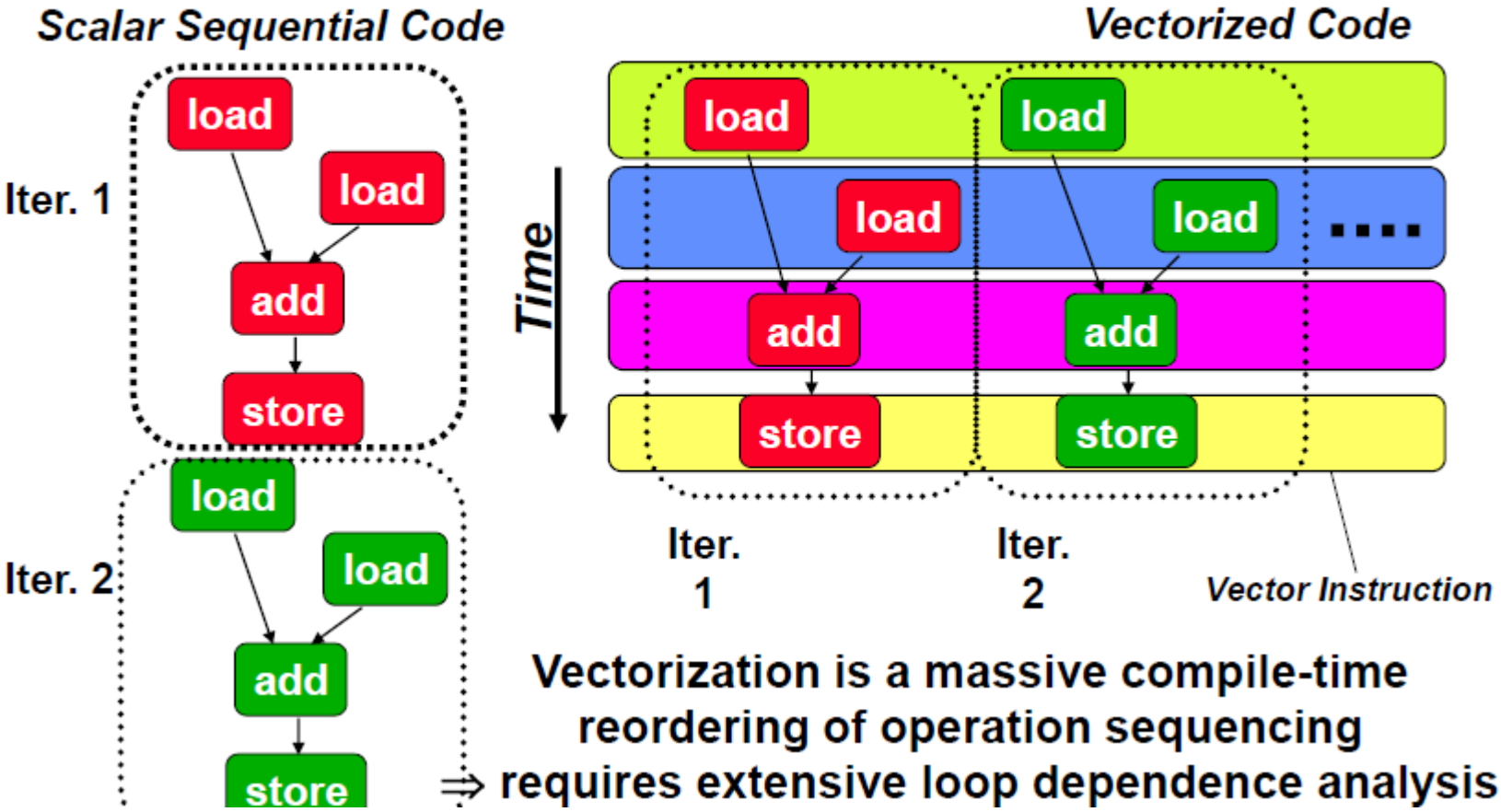
Optimizations (Cont')

How do you *program* a vector computer?

??

Automatic Code Vectorization

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



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

Programming Vector Architectures

- Compilers can provide feedback to programmers
- Programmers can provide hints to compiler

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid	Speedup from hint optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

Optimizations (Cont')

How does a vector processor handle programs where the vector lengths are not the same as the length of the vector register (64 for VMIPS)?

Most application vectors do not match the architecture vector length

Vector Length Register

```
for (i=0; i <n;  i=i+1)
    Y[i] = a * X[i] + Y[i];
```

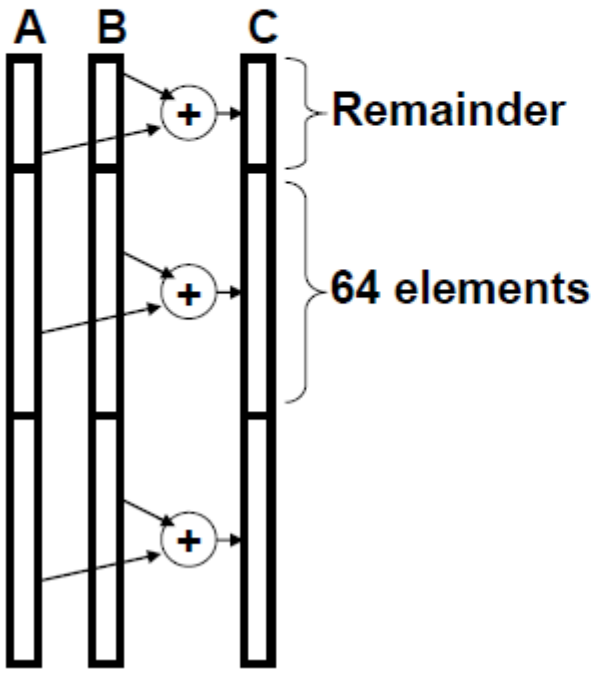
- Vector length not known at compile time?
- Use ***Vector Length Register (VLR)*** to control the length of any vector operation, including a vector load and store

Vector Stripmining

Problem: Vector registers have finite length

Solution: Break loops into pieces that fit into vector registers, *"Stripmining"*

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




```
        ANDI R1, N, 63     # N mod 64
        MTC1 VLR, R1       # Do remainder
loop:
        LV V1, RA
        DSLL R2, R1, 3     # Multiply by 8
        DADDU RA, RA, R2   # Bump pointer
        LV V2, RB
        DADDU RB, RB, R2
        ADDV.D V3, V1, V2
        SV V3, RC
        DADDU RC, RC, R2
        DSUBU N, N, R1 # Subtract elements
        LI R1, 64
        MTC1 VLR, R1    # Reset full length
        BGTZ N, loop    # Any more to do?
```

Optimizations (Cont')

What happens when there is an IF statement inside the code to be vectorized?

More code can vectorize if we can efficiently handle conditional statements

Vector Mask Registers

Problem: Want to vectorize loops with conditional code:

```
for (i=0; i<N; i++)
    if (A[i]>0) then
        A[i] = B[i];
```

Solution: Add vector *mask* (or *flag*) registers

- vector version of predicate registers, 1 bit per element

…and *maskable* vector instructions

- vector operation becomes NOP at elements where mask bit is clear

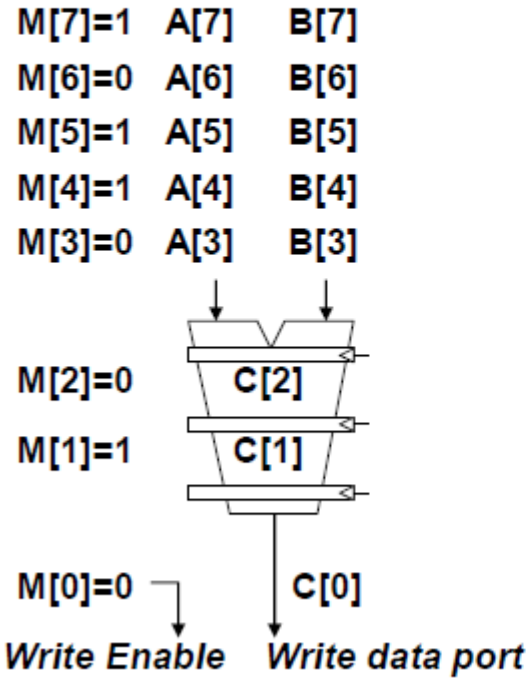
Code example:

```
CVM             # Turn on all elements
LV vA, rA       # Load entire A vector
SGTVS.D vA, F0  # Set bits in mask register where A>0
LV vA, rB       # Load B vector into A under mask
SV vA, rA       # Store A back to memory under mask
```

Masked Vector Instructions

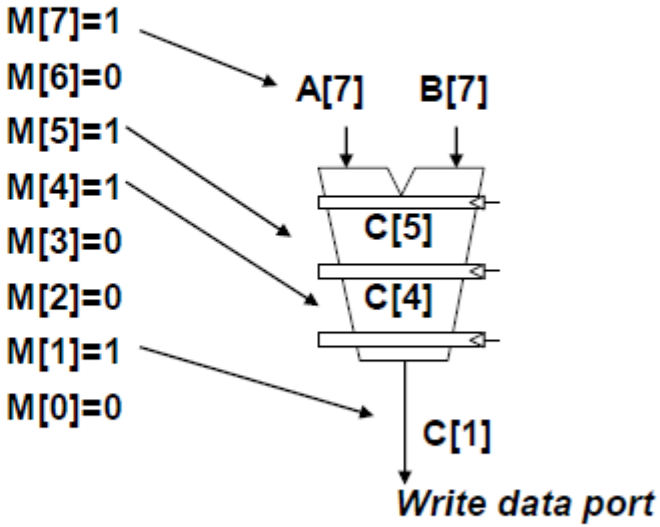
Simple Implementation

– 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



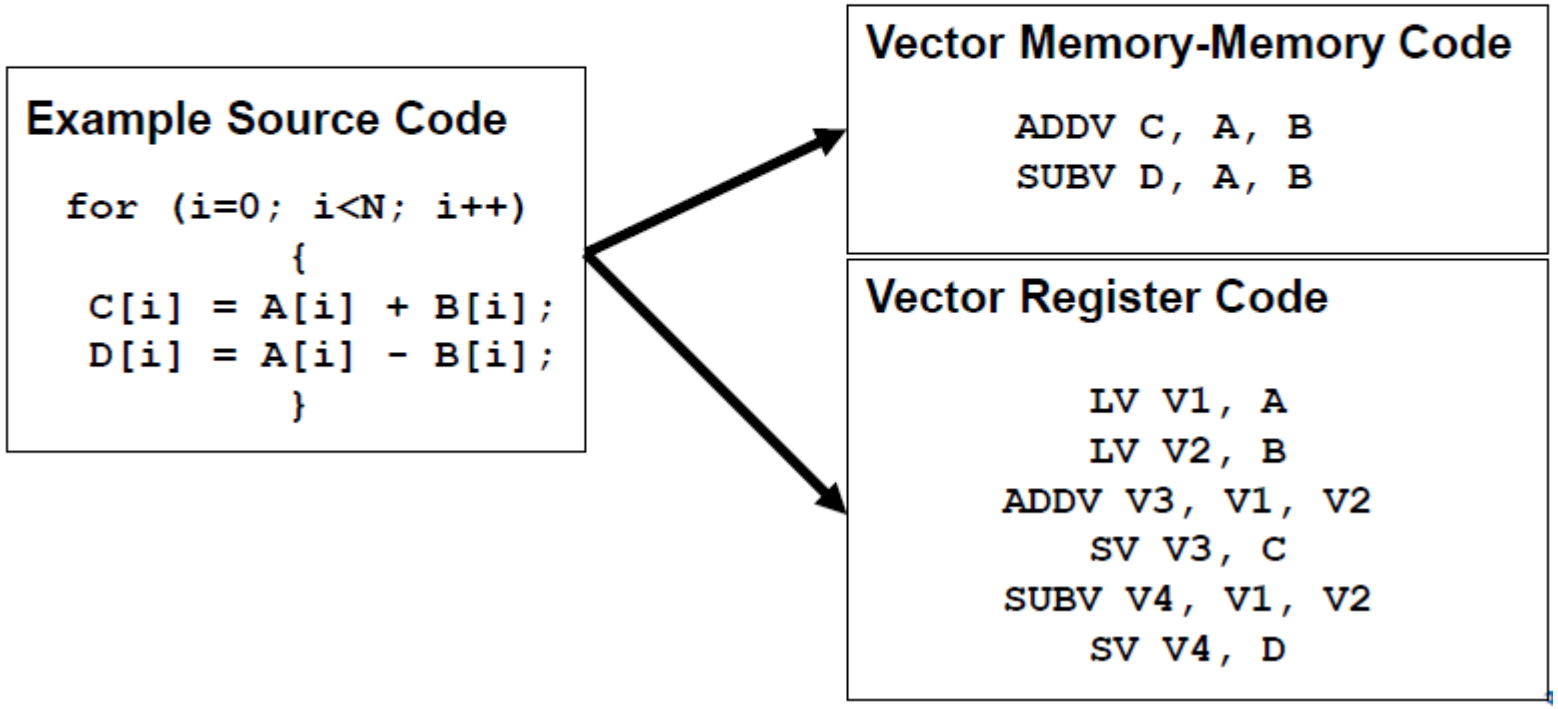
Optimizations (Cont')

What does a vector processor need from the *memory system*?

Without sufficient memory bandwidth, vector execution can be futile

Vector Memory-Memory vs. Vector Register Machines

- Vector memory-memory instructions hold all vector operands in main memory
- The first vector machines, CDC Star-100 ('73) and TI ASC ('71), were memory-memory machines
- Cray-1 ('76) was first vector register machine



Vector Memory-Memory vs. Vector Register Machines

- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
  - All operands must be read in and out of memory

Memory Banks

Supplying Bandwidth for Vector Load/Store Units

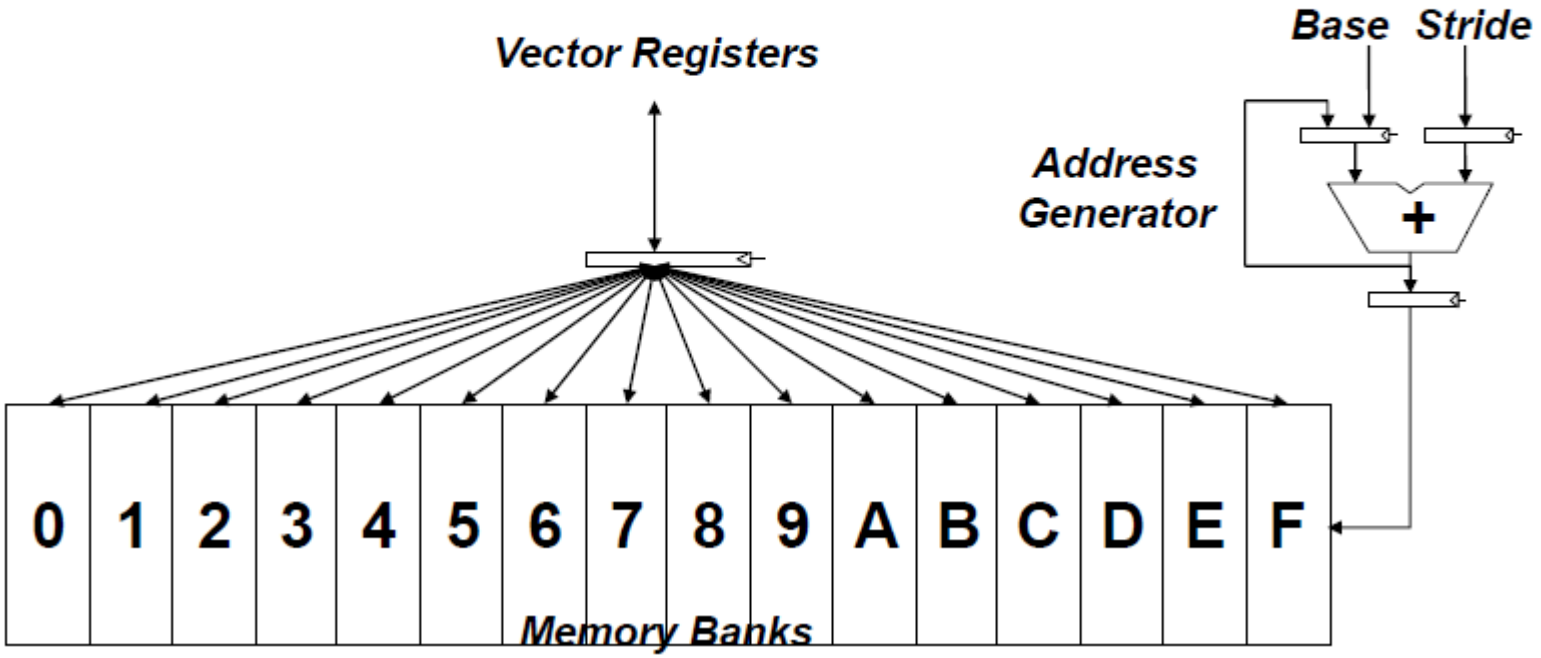
- Most vector processors use *memory banks*, which allow multiple independent accesses rather than simple memory interleaving for three reasons
  - **To support simultaneous accesses from multiple loads or stores**, the memory system needs multiple banks and to be able to control the addresses to the banks independently.
  - Non-sequential data access
  - Same memory system shared by multiple processors (with independent streams of addresses)

- In combination, these features lead to a large number of independent memory banks (~thousands)

Vector Memory System

Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

- *Bank busy time*: Cycles between accesses to same bank

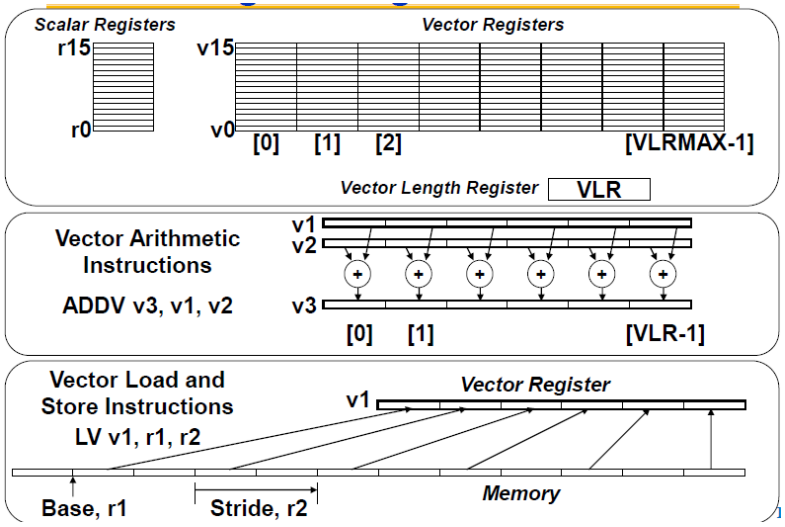


Optimizations (Cont')

How does a vector processor handle *multi-dimensional matrices*?

Matrices are popular

Stride



- Consider:

```
for (i = 0; i < 100; i=i+1)
        for (j = 0; j < 100; j=j+1) {
                A[i][j] = 0.0;
                for (k = 0; k < 100; k=k+1)
                A[i][j] = A[i][j] + B[i][k] * D[k][j];
        }
```

- Must vectorize multiplication of rows of B with columns of D
- Use *non-unit stride*

Optimizations (Cont')

How does a vector processor *handle sparse matrices*?

This is also popular data structure

Scatter-Gather

- Consider:

```
for (i = 0; i < n; i=i+1)
        A[K[i]] = A[K[i]] + C[M[i]];
```

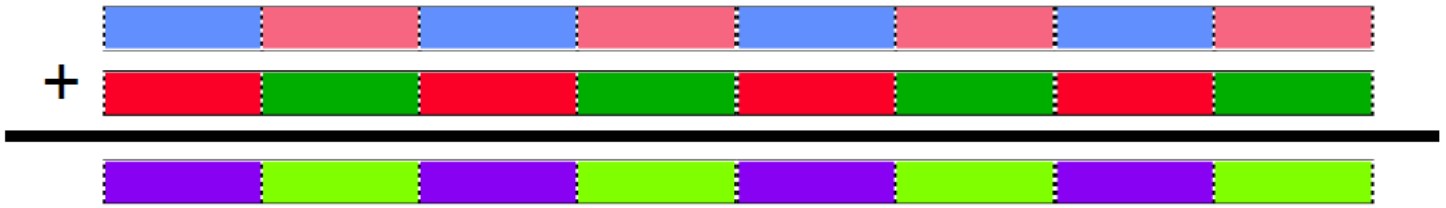
- Use ***index vectors*** K[] and M[]

```
LV        Vk, Rk          ;load K
LVI       Va, (Ra+Vk)     ;load A[K[]]
LV        Vm, Rm          ;load M
LVI       Vc, (Rc+Vm)     ;load C[M[]]
ADDVV.D   Va, Va, Vc      ;add them
SVI       (Ra+Vk), Va     ;store A[K[]]
```

Outline

- 4.1 Introduction
- 4.2 Vector Architecture
- 4.3 SIMD Instruction Set Extensions
- 4.4 GPU

SIMD Multimedia Extensions



- Media applications operate on data types narrower than the native word size
  - Example: disconnect ***carry chains*** to **"partition" adder**
- Support to handle short vectors added to existing ISAs
- Usually ***64-bit registers*** split into 2x32b or 4x16b or 8x8b
- Newer designs have ***256-bit registers***

MMX Instructions

- *Add, Subtract* in parallel: 8 8b, 4 16b, 2 32b
  - opt. signed/unsigned saturate (set to max) if overflow
- *Shifts (sll,srl, sra), And, And Not, Or, Xor* in parallel: 8 8b, 4 16b, 2 32b
- *Multiply* in parallel: 4 16b
- *Compare* = , > in parallel: 8 8b, 4 16b, 2 32b
  - sets field to 0s (false) or 1s (true); removes branches

Example SIMD Code for DAXPY

```
        L.D  F0,a                ;load scalar a
        MOV  F1, F0              ;copy a into F1 for SIMD MUL
        MOV  F2, F0              ;copy a into F2 for SIMD MUL
        MOV  F3, F0              ;copy a into F3 for SIMD MUL
        DADDIU  R4,Rx,#512       ;last address to load
Loop:   L.4D F4,0[Rx]            ;load X[i], X[i+1], X[i+2], X[i+3]
        MUL.4D F4,F4,F0          ;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
        L.4D    F8,0[Ry]         ;load Y[i], Y[i+1], Y[i+2], Y[i+3]
        ADD.4D F8,F8,F4          ;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
        S.4D    0[Ry],F8         ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
        DADDIU Rx,Rx,#32         ;increment index to X
        DADDIU Ry,Ry,#32         ;increment index to Y
        DSUBU  R20,R4,Rx         ;compute bound
        BNEZ    R20,Loop         ;check if done
```

Why are Multimedia SIMD instructions so popular?

- Cost little to add to the standard arithmetic unit
- Require little extra state compared to vector architectures
- Need a lot of memory bandwidth to support a *vector architecture*, which many computers don't have
- Do not have to deal with problems in virtual memory when a single instruction that can generate 64 memory accesses can get a page fault in the middle of the vector

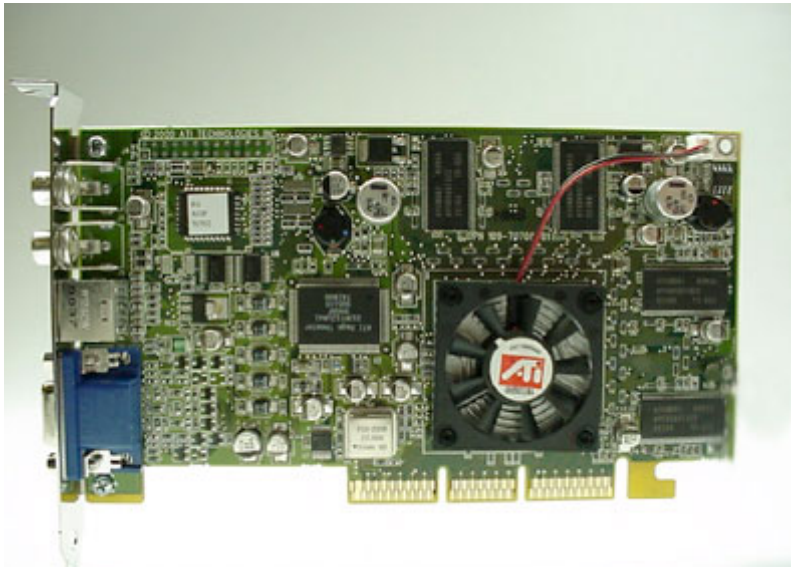
Three major omissions in SIMD vs. vector

- Limited instruction set:
  - Fixed number of data operands in opcode
  - no vector length control (several fixed lengths)
  - no strided load/store or scatter/gather
- Limited vector register length:
  - requires superscalar dispatch to keep multiply/add/load units busy
- Limited mask registers:
  - to support conditional execution of elements as in vector processors

Outline

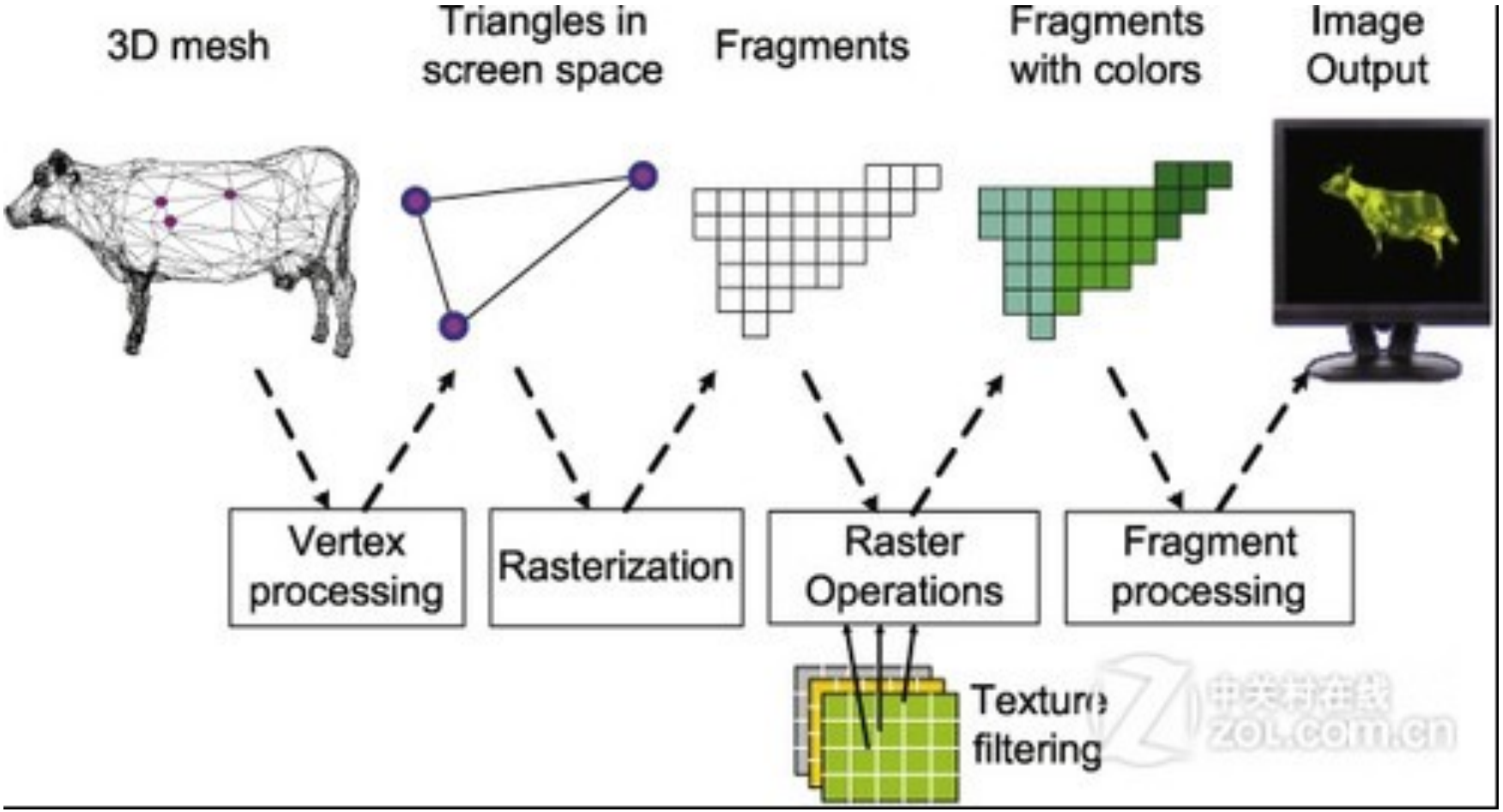
- 4.1 Introduction
- 4.2 Vector Architecture
- 4.3 SIMD Instruction Set Extensions
- 4.4 GPU (Graphic Processing Unit)

What is a Graphics Card?

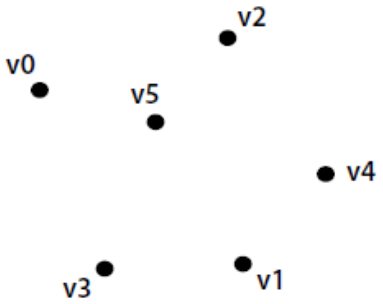


Graphics cards controls what is to be shown on a computer monitor and calculates 3D images and graphics.

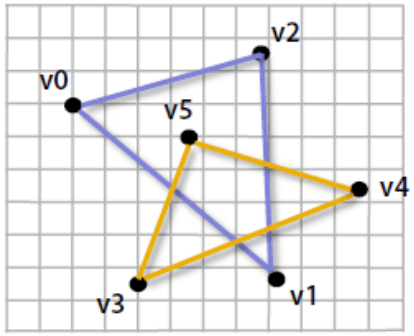
Main Steps


3D mesh
Triangles in screen space
Fragments
Fragments with colors
Image Output
Vertex processing
Rasterization
Raster Operations
Fragment processing
Texture filtering

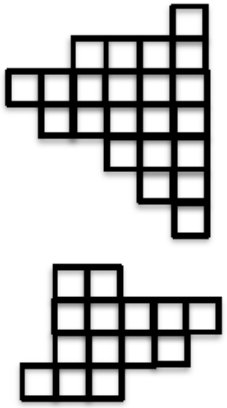
Main Steps



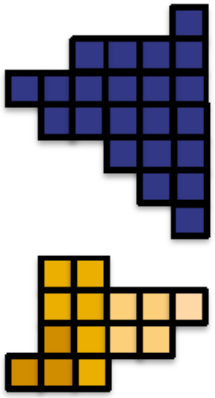
Vertices



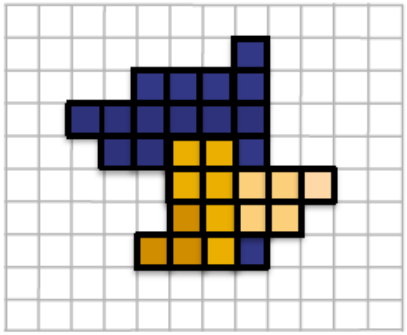
Primitives



Fragments

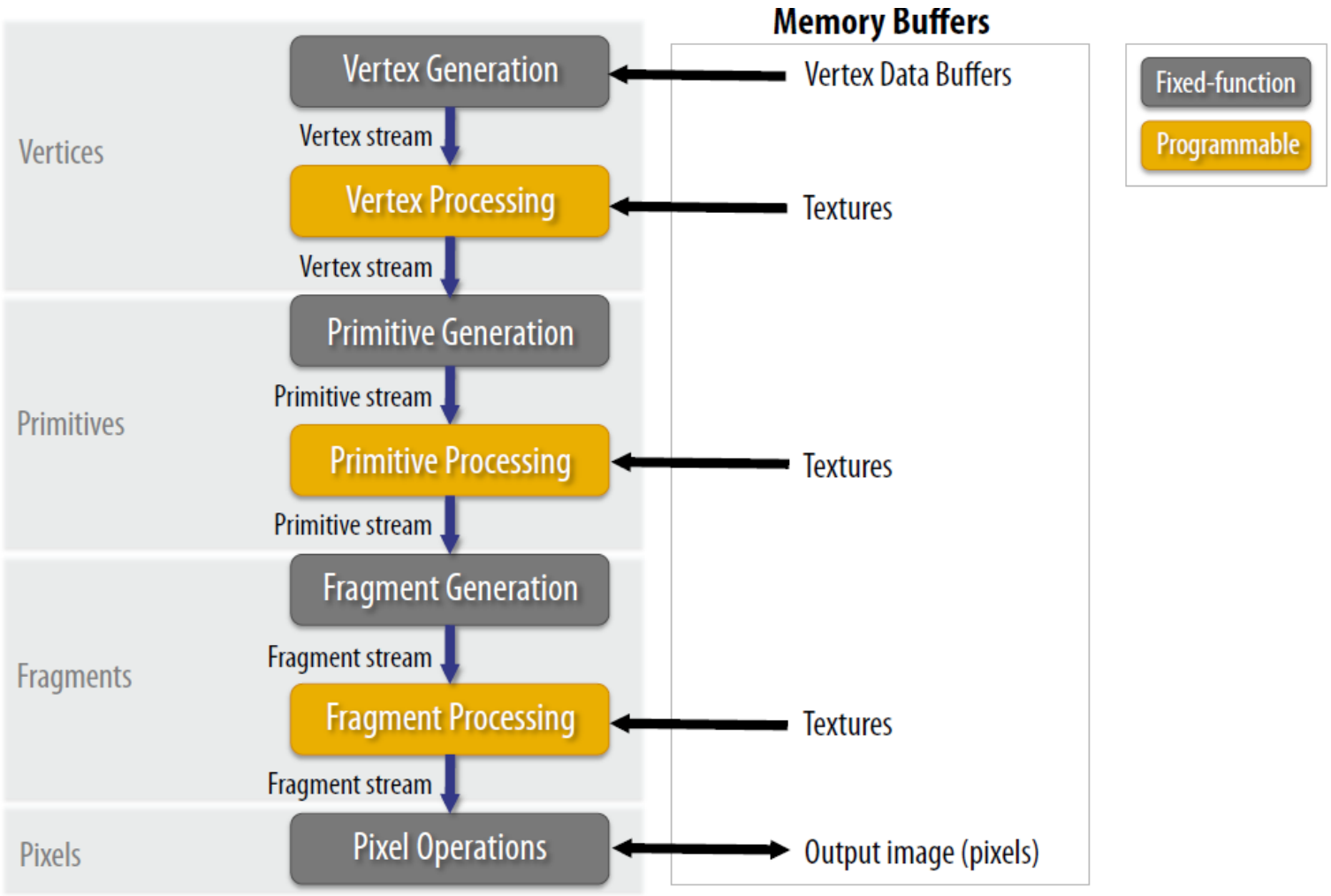


Fragments (shaded)



Pixels

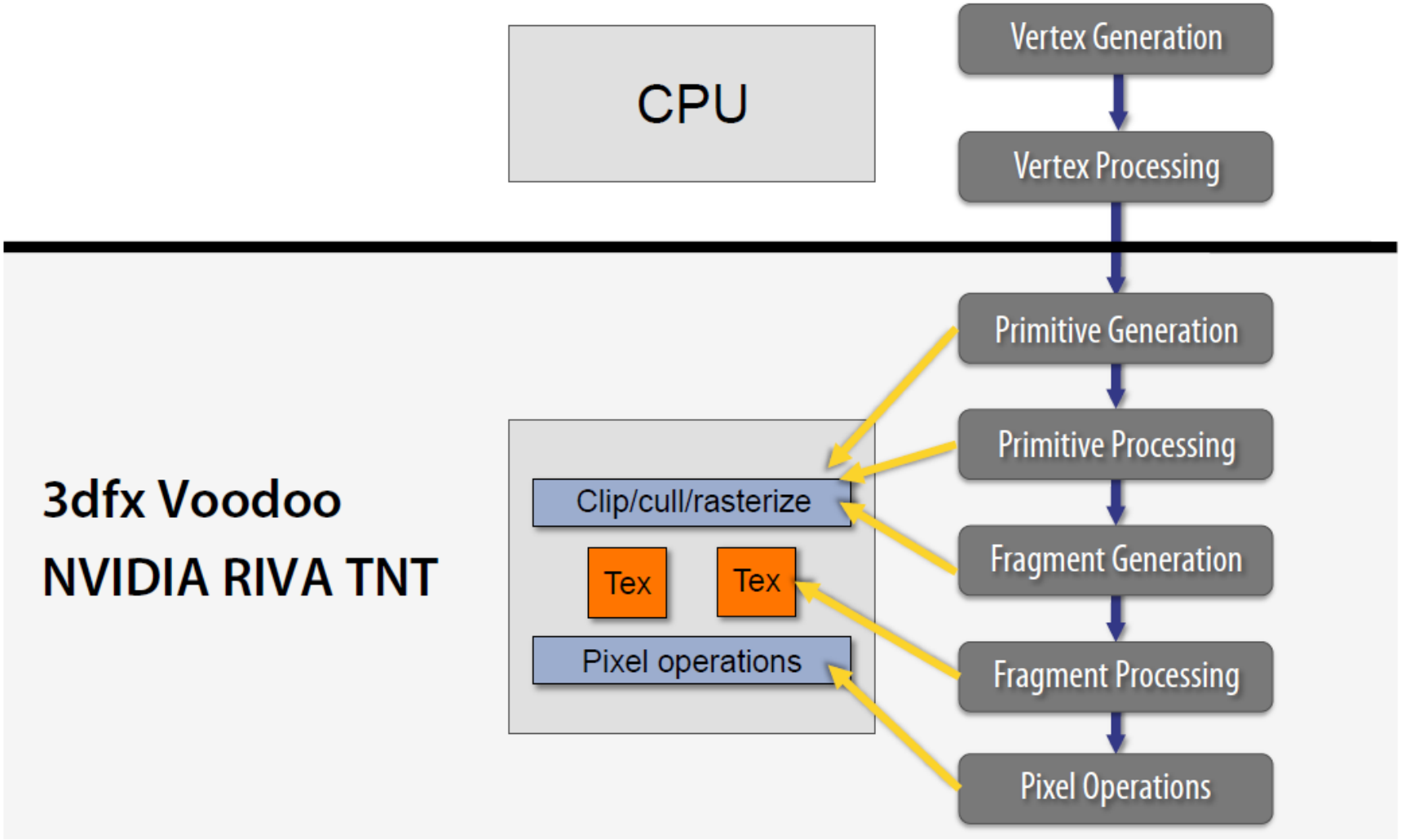
Graphics Pipeline



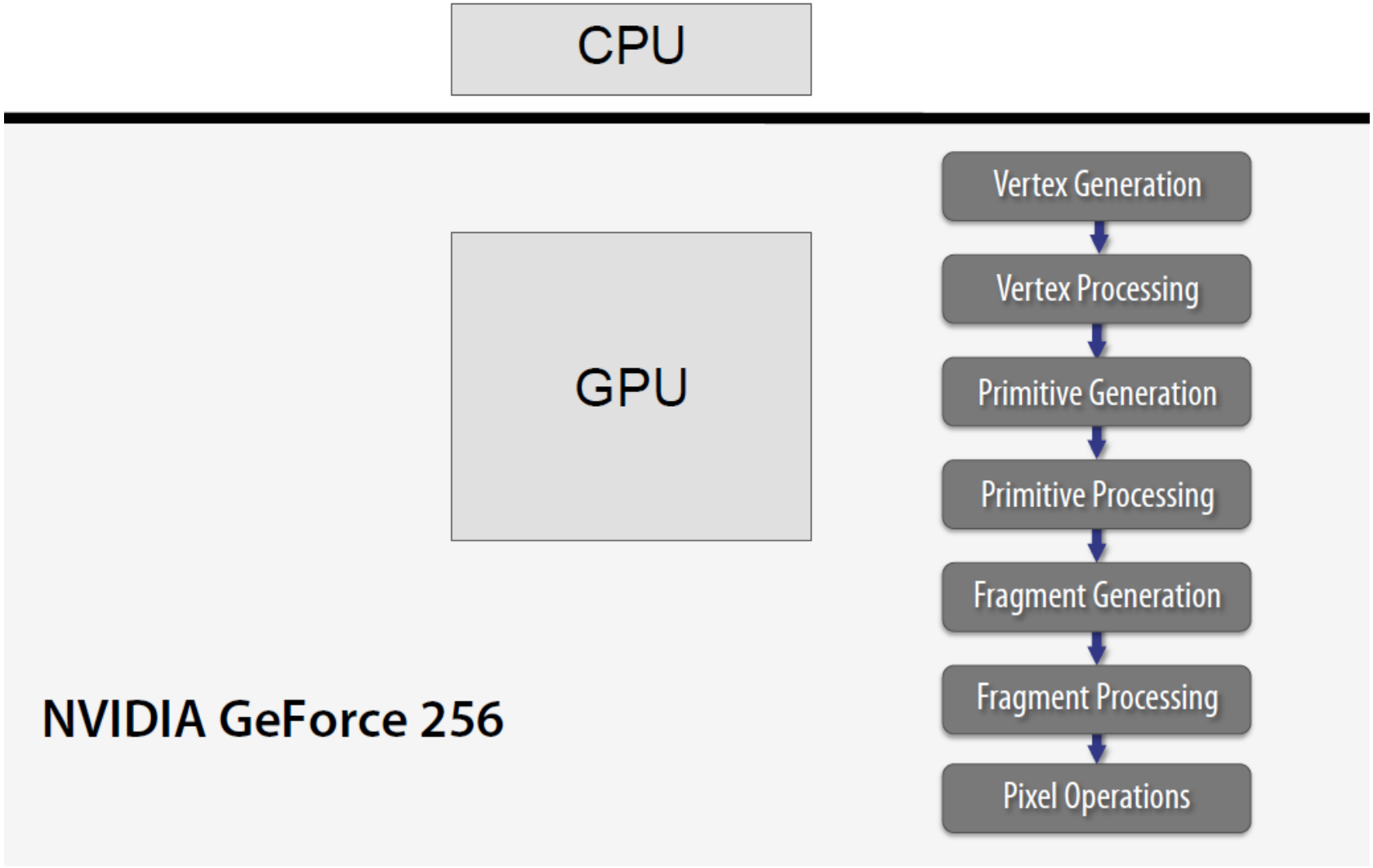
GPU Evolvement

- Original GPUs were dedicated fixed-function devices for generating 3D graphics (mid-late 1990s) including high-performance floating-point units
  - *Programmability was an afterthought*
  - *Started from 1999, GeForce 256*
- Over time, more programmability added (2001-2005)
  - New language Cg (Nvidia) for writing small programs run on each vertex or each pixel
- Some users noticed they could do general-purpose computation by mapping input and output data to images, and computation to vertex and pixel shading computations
  - Incredibly difficult programming model as had to use graphics pipeline model for general computation

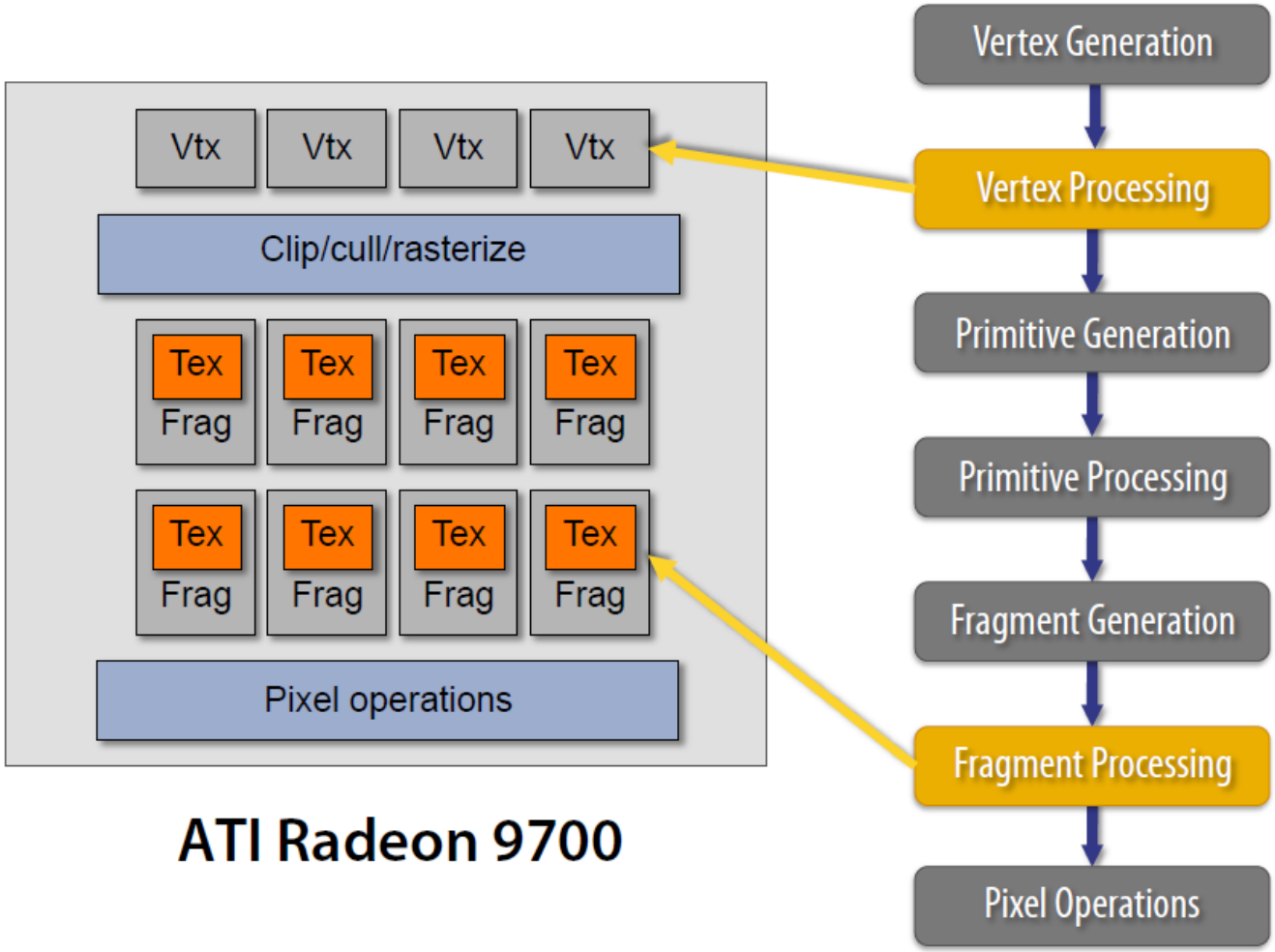
Pre-1999 PC 3D graphics accelerator



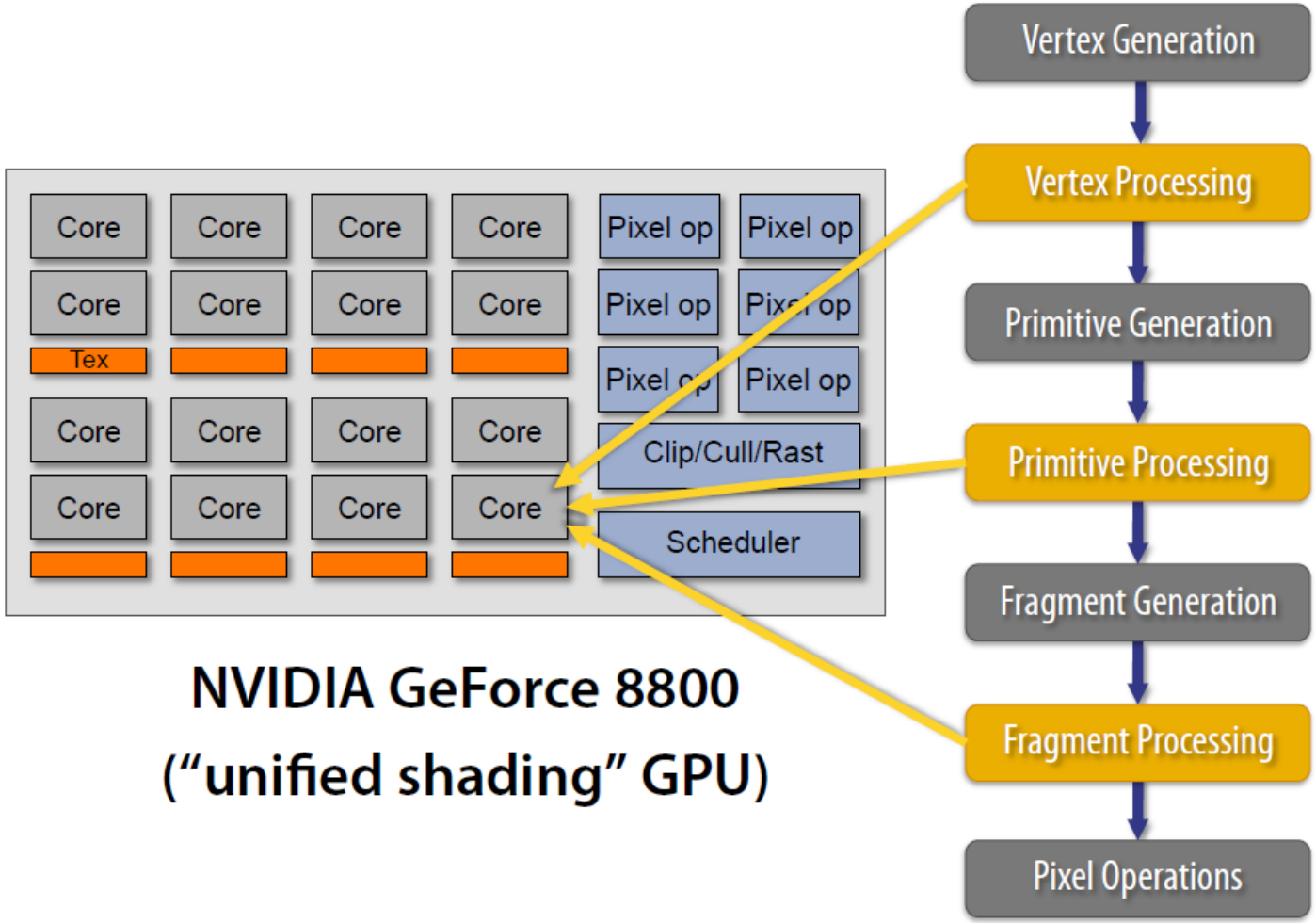
GPU* circa 1999



Direct3D 9 programmability: 2002



Direct3D 10 programmability: 2006

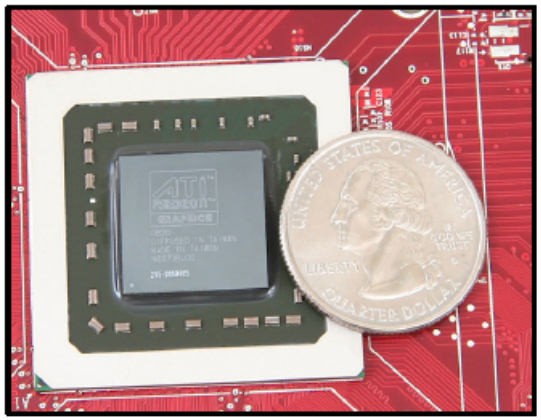


GPU is fast!

Intel Core i7 Quad

~100 GFLOPS peak
730 million transistors

(obtainable if you code your program to use 4 threads and SSE vector instr)

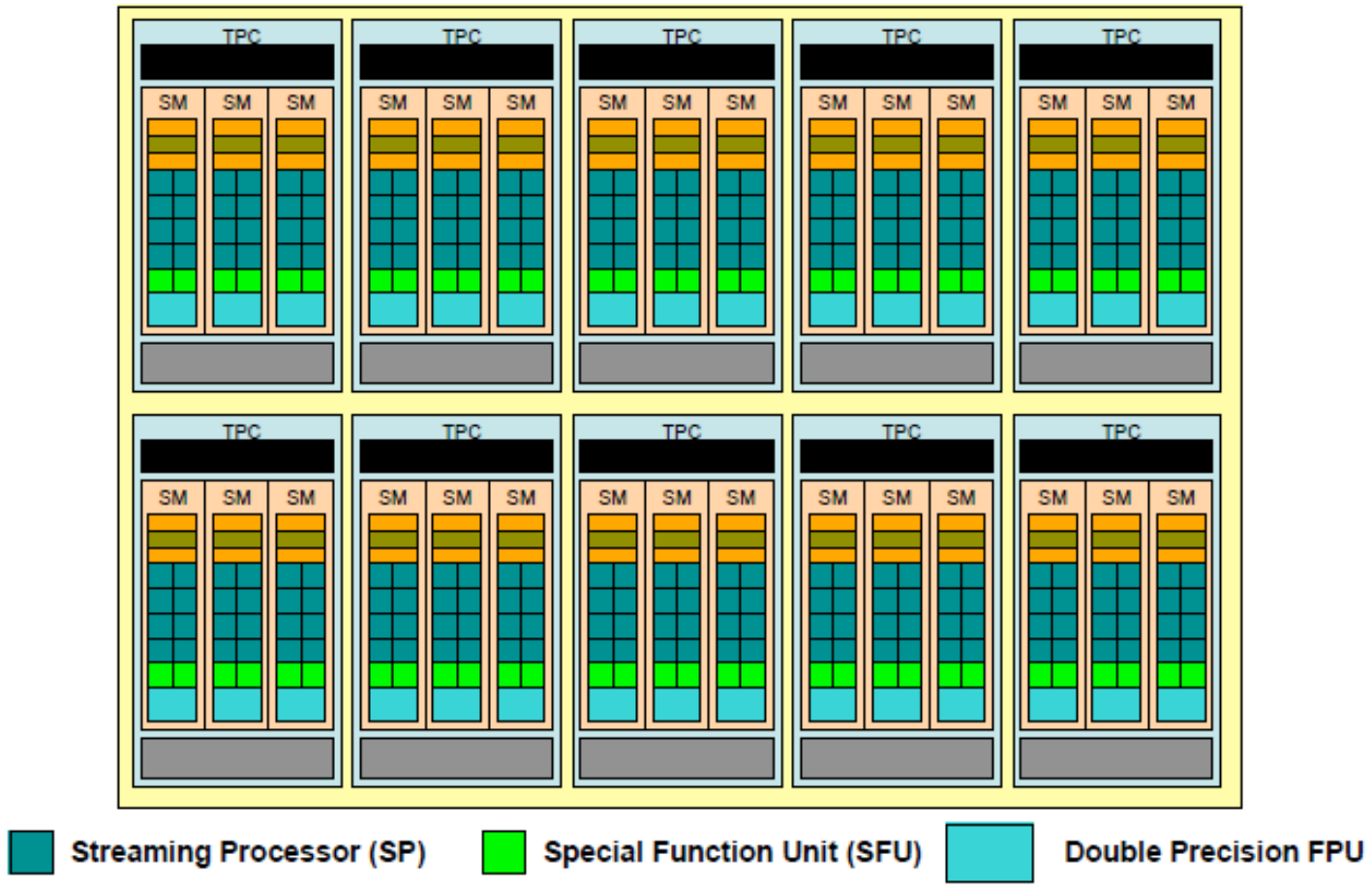


AMD Radeon HD 5870

~2.7 TFLOPS peak
2.2 billion transistors

(obtainable if you write OpenGL programs

Large Number of Cores



NVIDIA Tesla C870

- 518 Gflops/card
- 1.5 GB Memory
- 128 SM processors

Building Blocks for Supercomputers

Rank	Site	Computer
1	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu
2	National Supercomputing Center in Tianjin China	NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050 NUDT
3	DOE/SC/Oak Ridge National Laboratory United States	Cray XT5-HE Opteron 6-core 2.6 GHz Cray Inc.
4	National Supercomputing Centre in Shenzhen (NSCS) China	Dawning TC3600 Blade System, Xeon X5650 6C 2.66GHz, Infiniband QDR, NVIDIA 2050 Dawning
5	GSIC Center, Tokyo Institute of Technology Japan	HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows NEC/HP
6	DOE/NNSA/LANL/SNL United States	Cray XE6, Opteron 6136 8C 2.40GHz, Custom Cray Inc.
7	NASA/Ames Research Center/NAS United States	SGI Altix ICE 8200EX/8400EX, Xeon HT QC 3.0/Xeon 5570/5670 2.93 Ghz, Infiniband SGI
8	DOE/SC/LBNL/NERSC United States	Cray XE6, Opteron 6172 12C 2.10GHz, Custom Cray Inc.
9	Commissariat a l'Energ Atomique (CEA) France	
10	DOE/NNSA/LANL United States	BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband IBM

www.top500.org
Nov. 2011

Product Description ENLARGE

The NVIDIA Tesla C2050 brings computing and performance of a small cluster to the desktop. Based on the CUDA architecture, the C2050 delivers supercomputing power at 1/20th the power consumption and 1/10th the cost.

Features

448 Processing Cores; 3GB 384-bit GDDR5; PCI Express 2.0 x16; NVIDIA CUDA

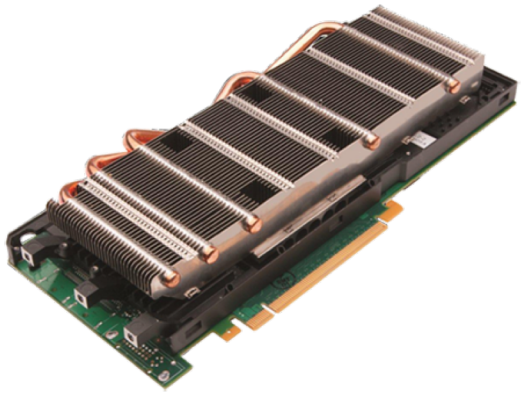
Building Blocks for Supercomputers

TOP 10 Sites for November 2013

For more information about the sites and systems in the list, click on the links or view the complete list.

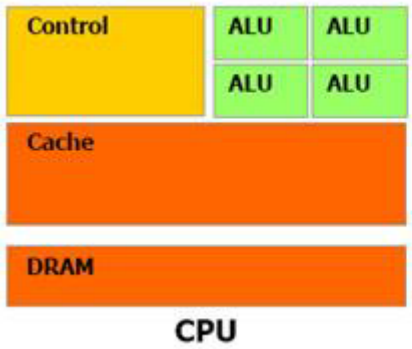
Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325
7	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462,462	5,168.1	8,520.1	4,510
8	Forschungszentrum Juelich (FZJ) Germany	**JUQUEEN** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	458,752	5,008.9	5,872.0	2,301
9	DOE/NNSA/LLNL United States	**Vulcan** - BlueGene/ Interconnect IBM				
10	Leibniz Rechenzentrum Germany	**SuperMUC** - iDataP Infiniband FDR IBM				

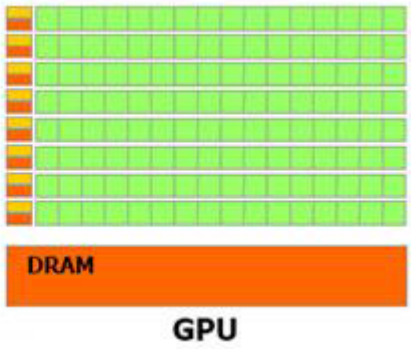
www.top500.org
Nov. 2013

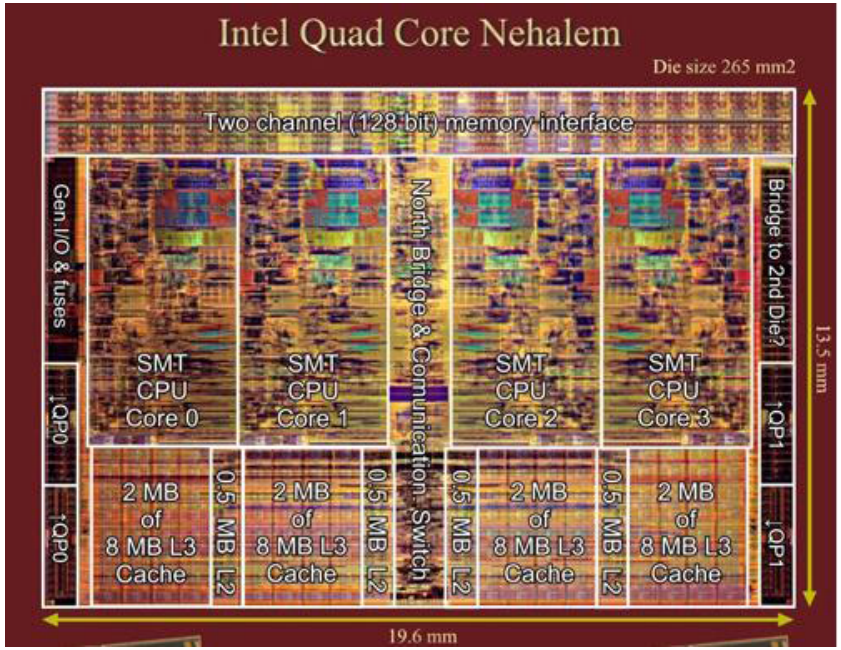


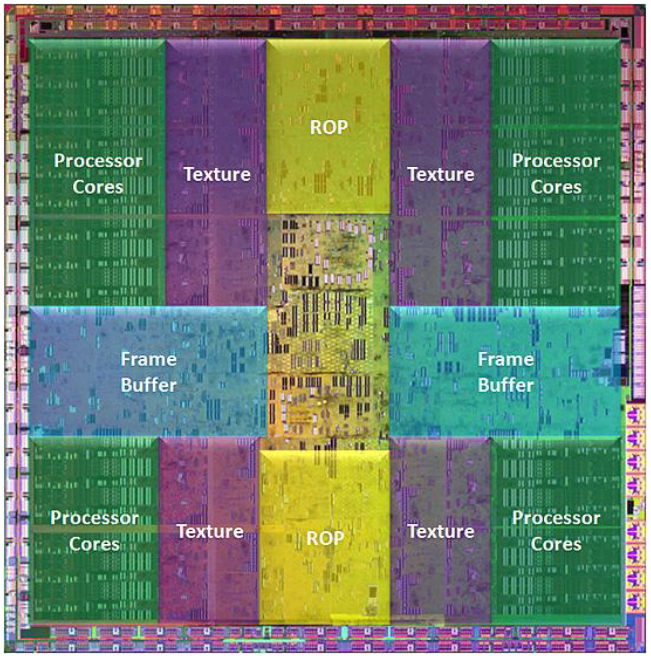
TECHNICAL SPECIFICATIONS	TESLA K20X
Peak double precision floating point performance (board)	1.31 teraflops
Peak single precision floating point performance (board)	3.95 teraflops
Number of GPUs	
Number of CUDA cores	2688
Memory size per board (GDDR5)	6 GB
Memory bandwidth for board (ECC off)[b]	250 GBytes/sec
GPU computing applications	ıg, computational chemistry satellite imaging, weather
Architecture features	łyper-Q
System	Servers only

Comparing CPU and GPU









General-Purpose GPUs (GP-GPUs)

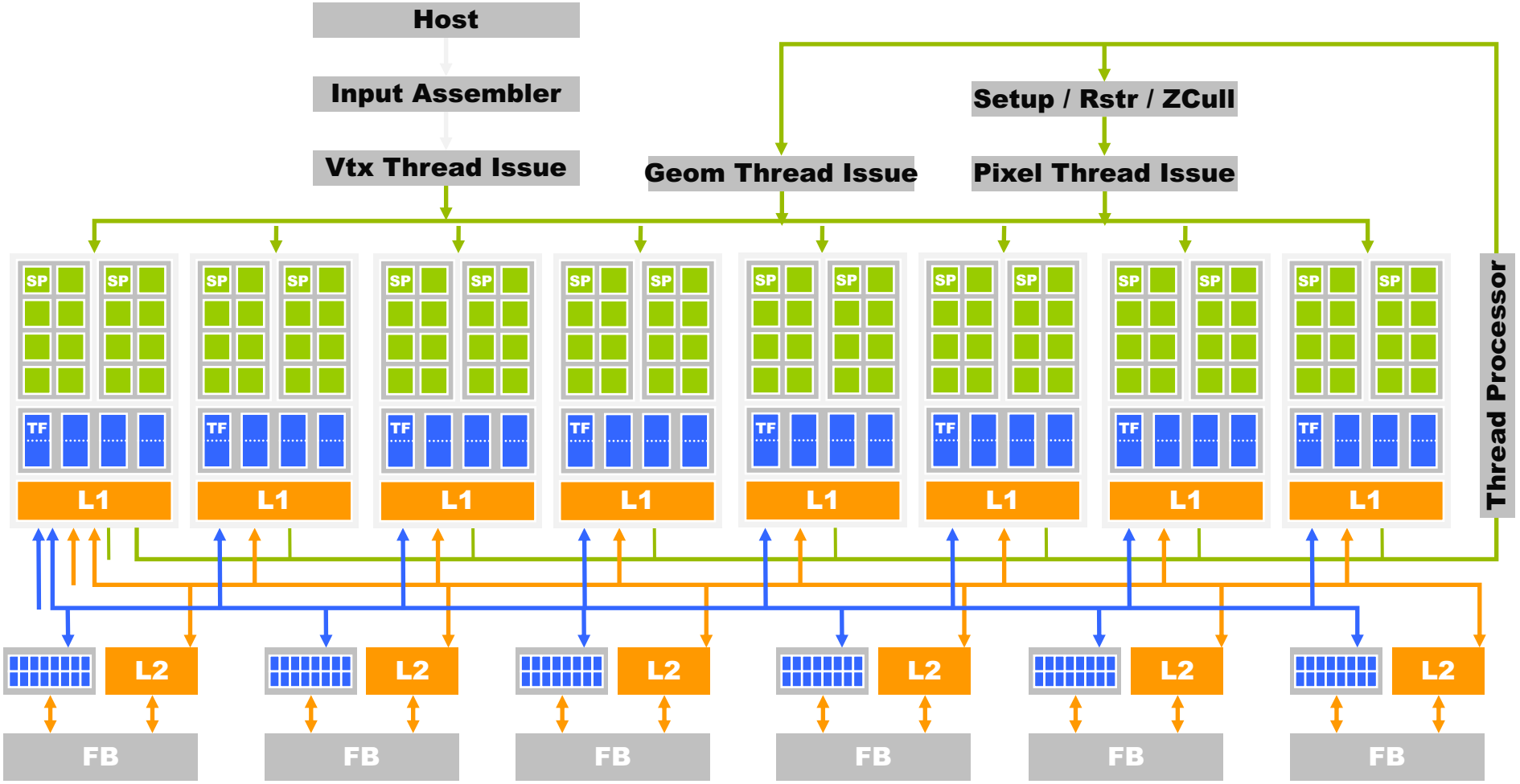
- **Idea: Take advantage of GPU *computational performance* and *memory bandwidth* to accelerate some kernels for general-purpose computing**
- **Host CPU issues data-parallel kernels to GP-GPU for execution**
- Programming model is "Single Instruction Multiple Thread (SIMD/SIMT)"

CUDA

Compute Unified Device Architecture

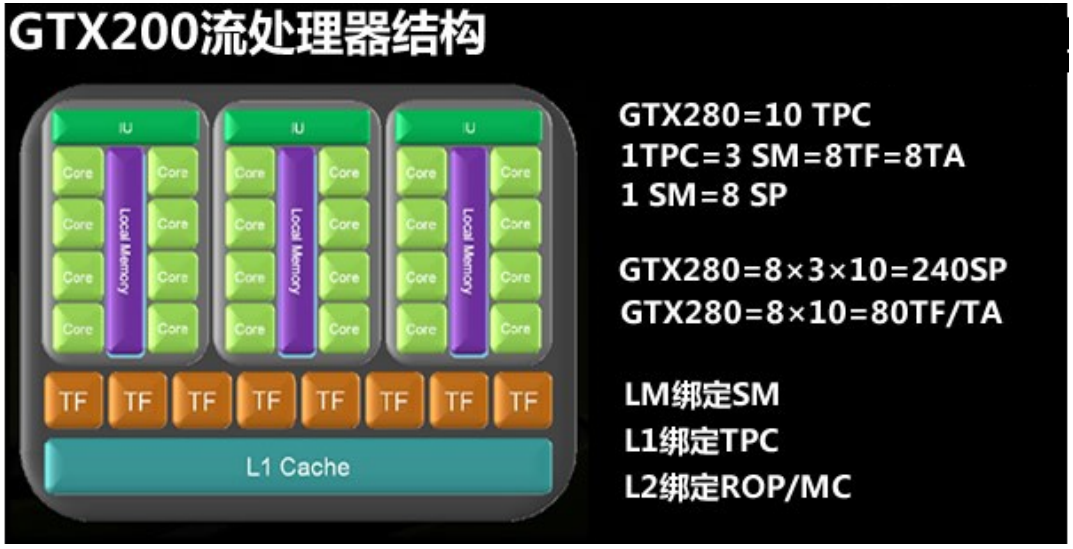
- **In 2006, Nvidia introduced GeForce 8800 GPU supporting *a new programming language*: CUDA**
- CUDA™ is a *parallel computing platform* and *programming model* invented by NVIDIA.
- It enables *dramatic increases in computing performance* by harnessing the power of the graphics processing unit (GPU).

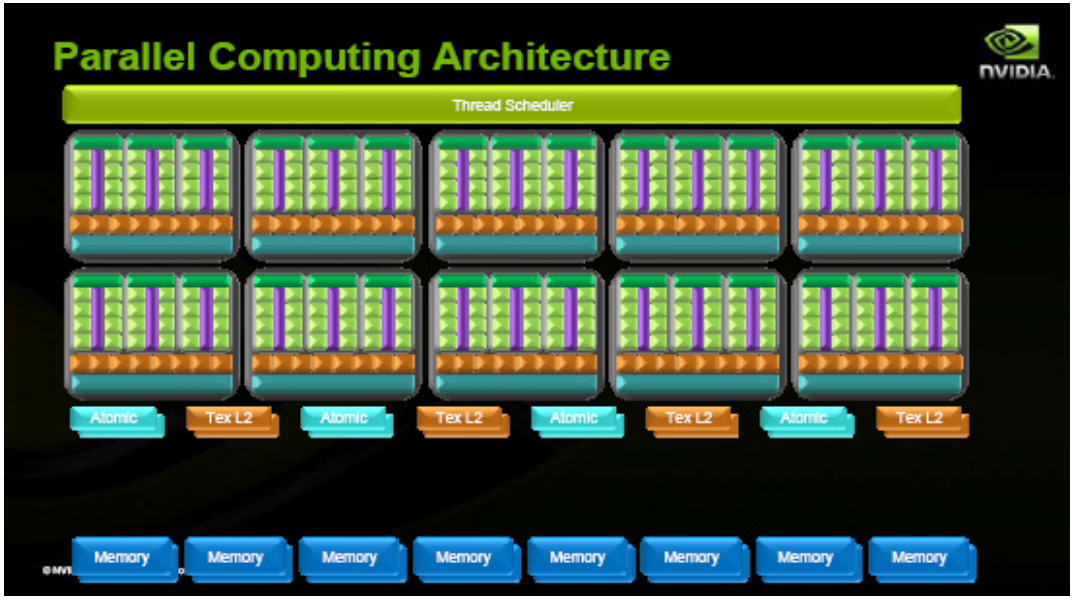
GPU Hardware (e.g., G80 GPU)



GPU Hardware (NVIDIA GTX200)

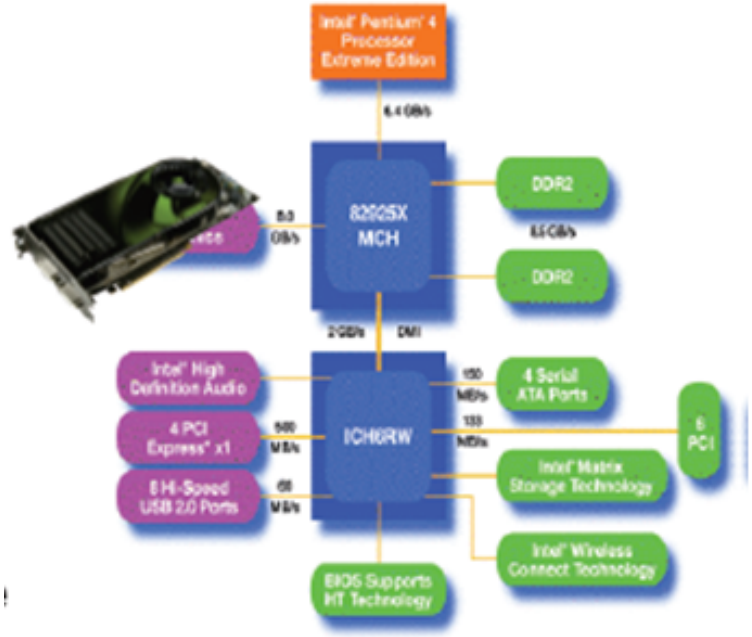
- GTX200 10 TPC（Texture Processing Cluster）
- Each TPC consists of （Streaming Multiprocessors）
- Each SM consists of SP （Stream Processor）
- Each SM supports up to 1024 threads





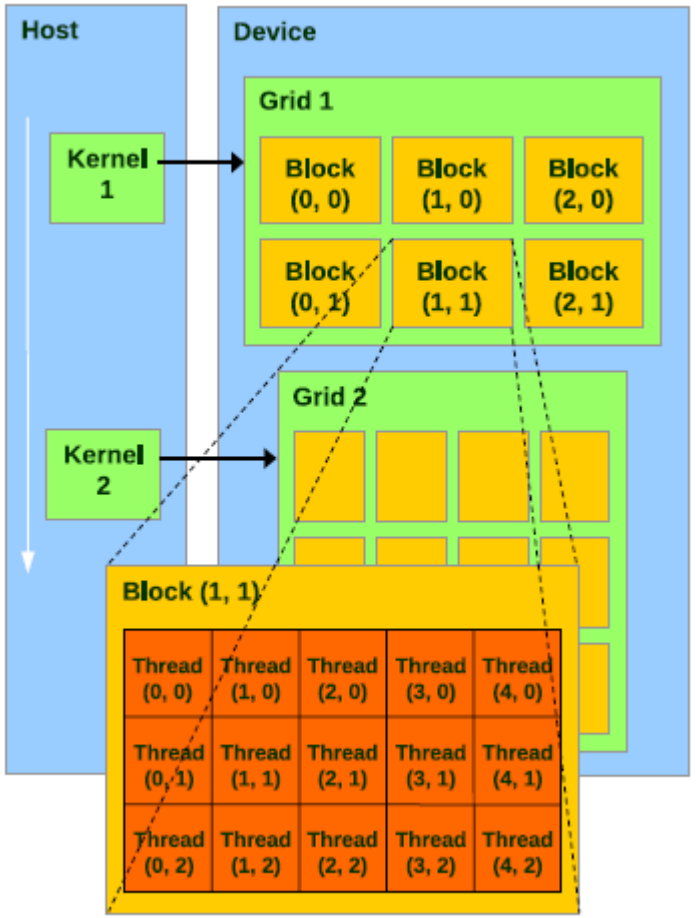
CUDA: Heterogeneous Computing

- CPU
  - Serial code
- GPU
  - Highly parallel

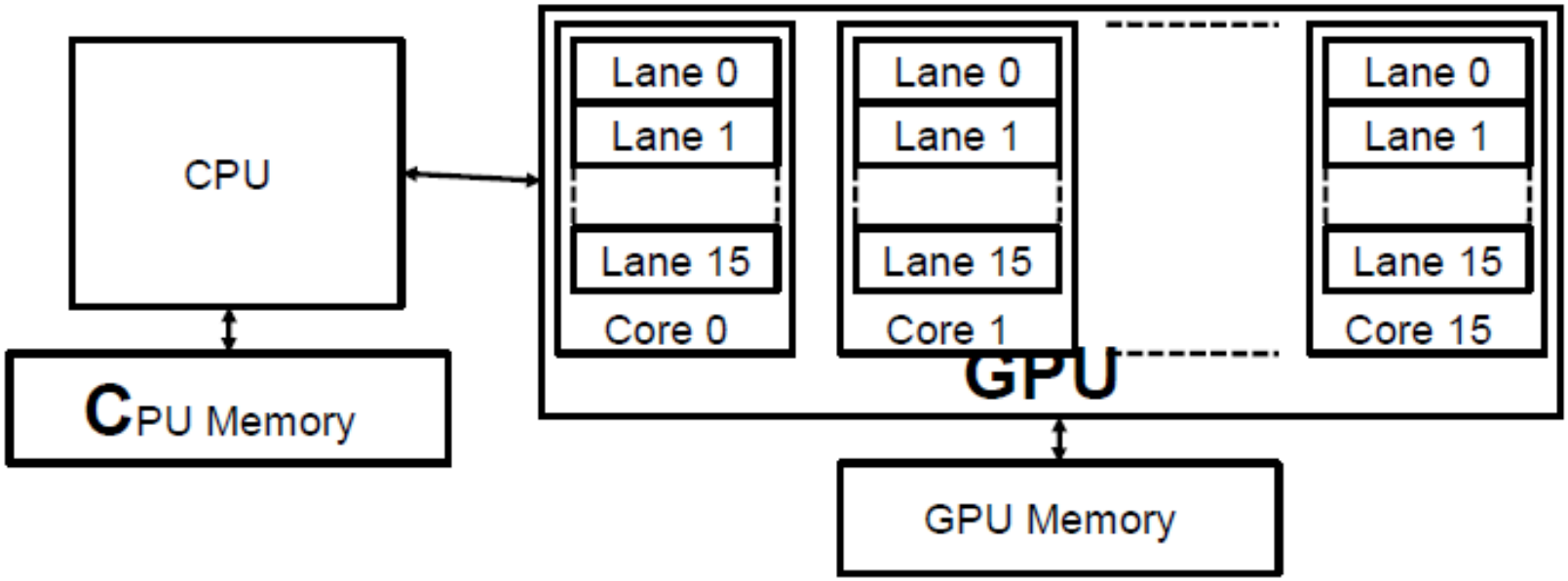


Overview of CUDA Programming

- A kernel is executed as a Grid of thread Blocks
  - Up to *512 threads* in one block
  - All threads in a block execute the same program (kernel) but on different data
  - A block is assigned to a processor that executes the code
  - Independent blocks
- Only ONE kernel at a time



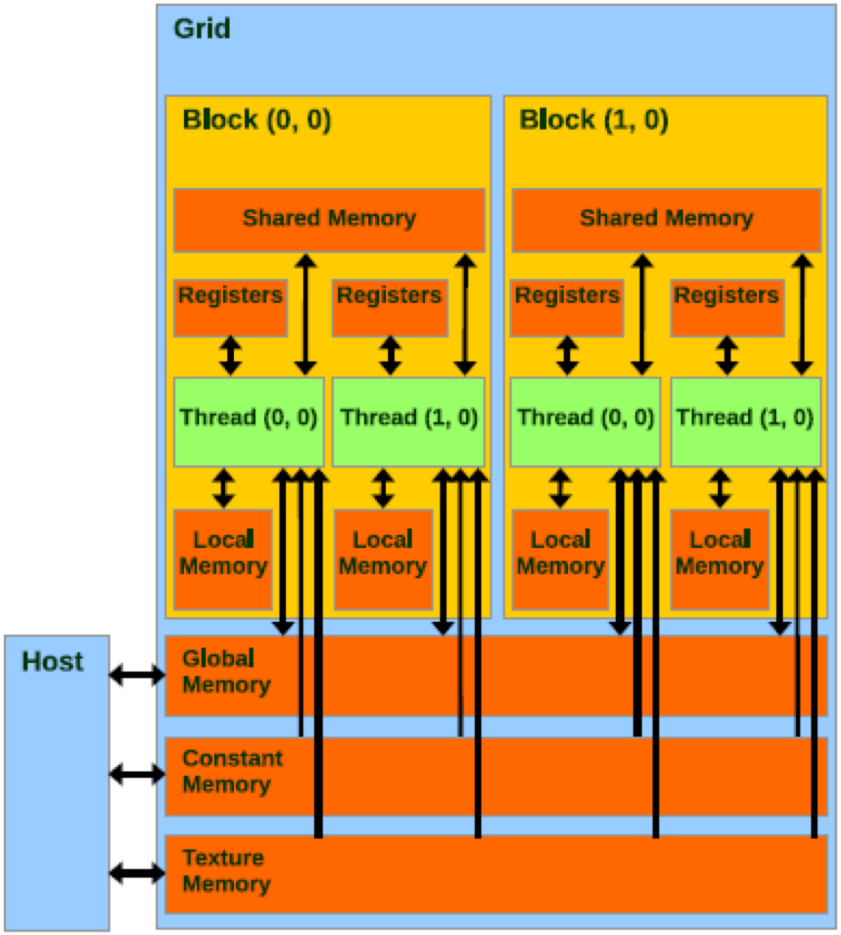
Hardware Execution Model



- GPU is built *from multiple parallel cores*, each core contains a *multithreaded SIMD processor* with multiple lanes but with no scalar processor
- CPU sends whole "grid" to GPU, which distributes *thread blocks* among *cores* (each thread block executes on one core)

Memory Hierarchy

- Registers
- Local memory
- Shared memory (shared by different threads in same block)
- Device memory (texture, constant, local, global)

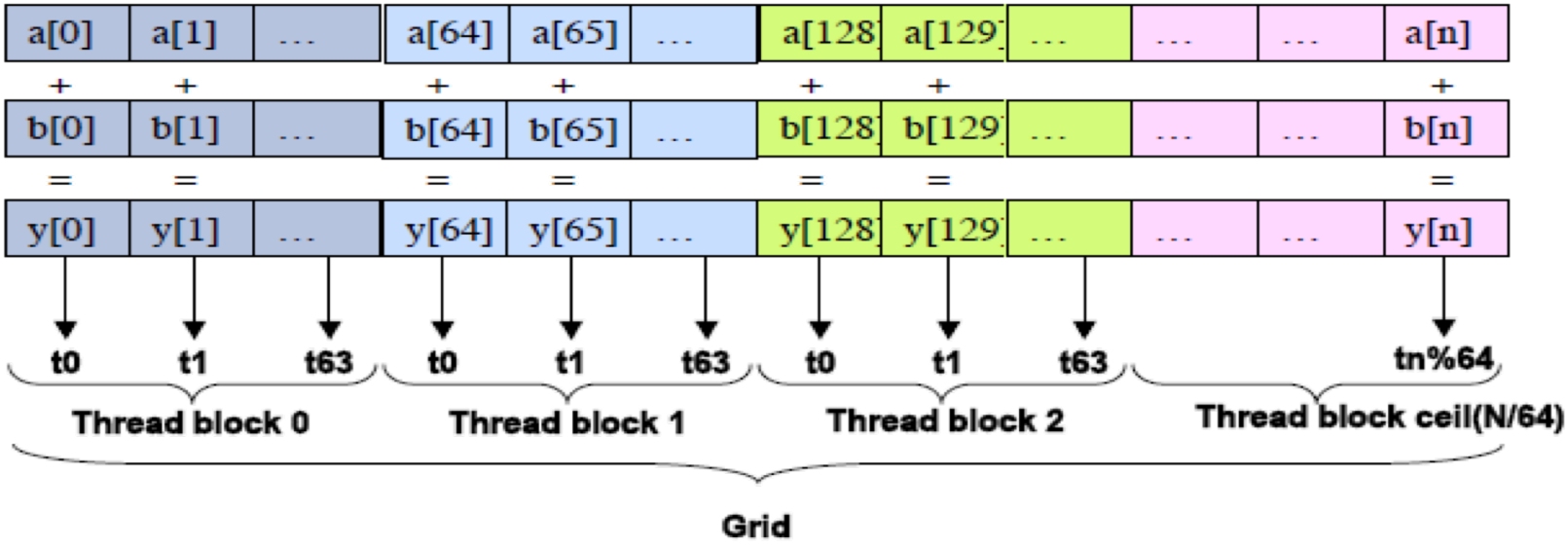


CUDA Memory Types

registers	read-write per-thread
local memory	read-write per-thread
shared memory	read-write per-block
global memory	read-write per-grid
constant memory	read-only per-grid
texture memory	read-only per-grid

Example for Data Division and Thread

- Vector addition (N elements/vector)
  - 1 thread for one addition
  - 64 threads per block
  - ceil(N/64) thread blocks



Example: DAXPY

```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
            for (int i = 0; i < n; ++i)
                  y[i] = a*x[i] + y[i];
}
```

Conventional C code for the DAXPY loop!

CUDA code for DAXPY

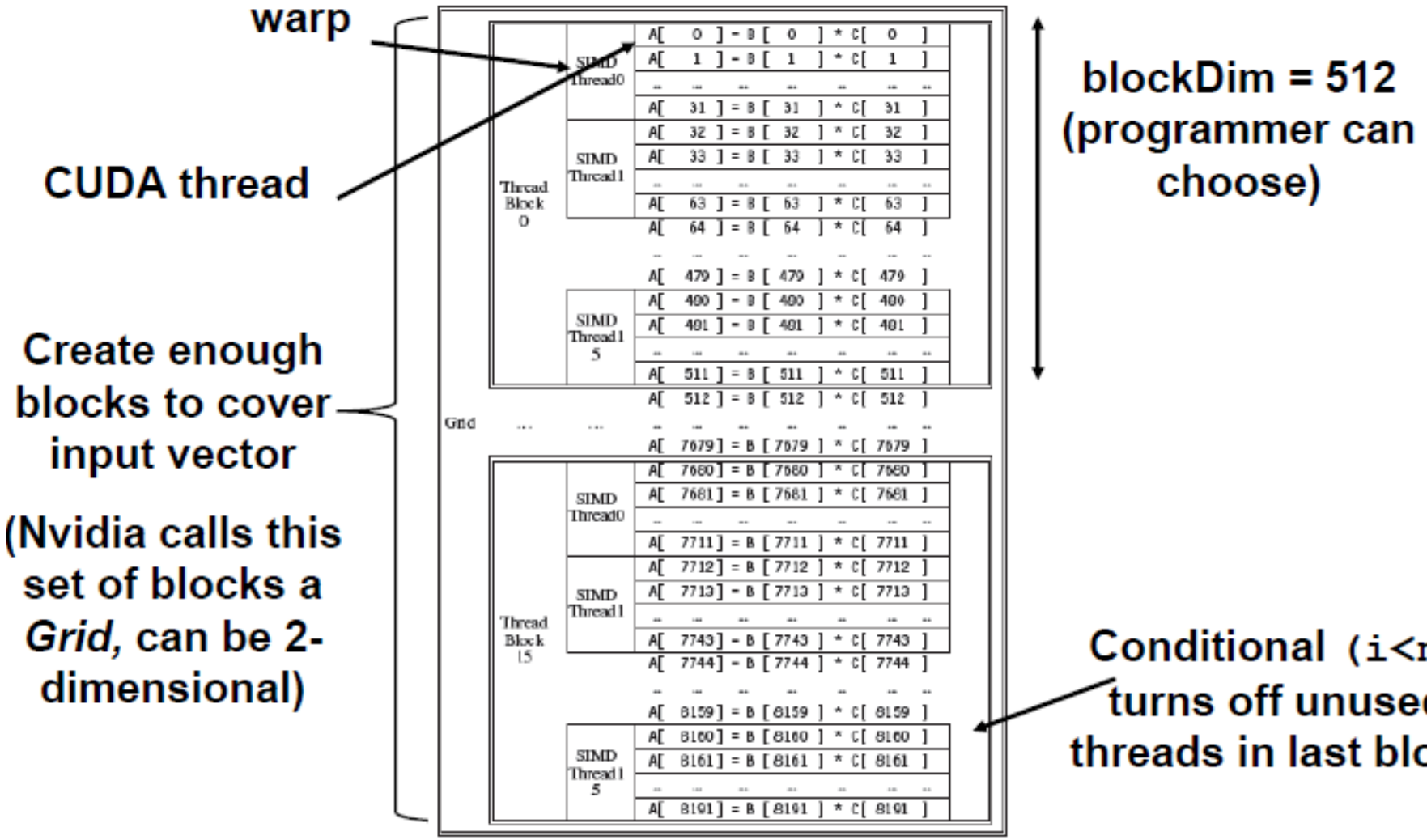
```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+ 255) / 256;
   daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if (i < n) y[i] = a*x[i] + y[i];
}
```

- Launch *n* threads, one per element
- 256 CUDA threads per thread block in a multithreaded SIMD processor

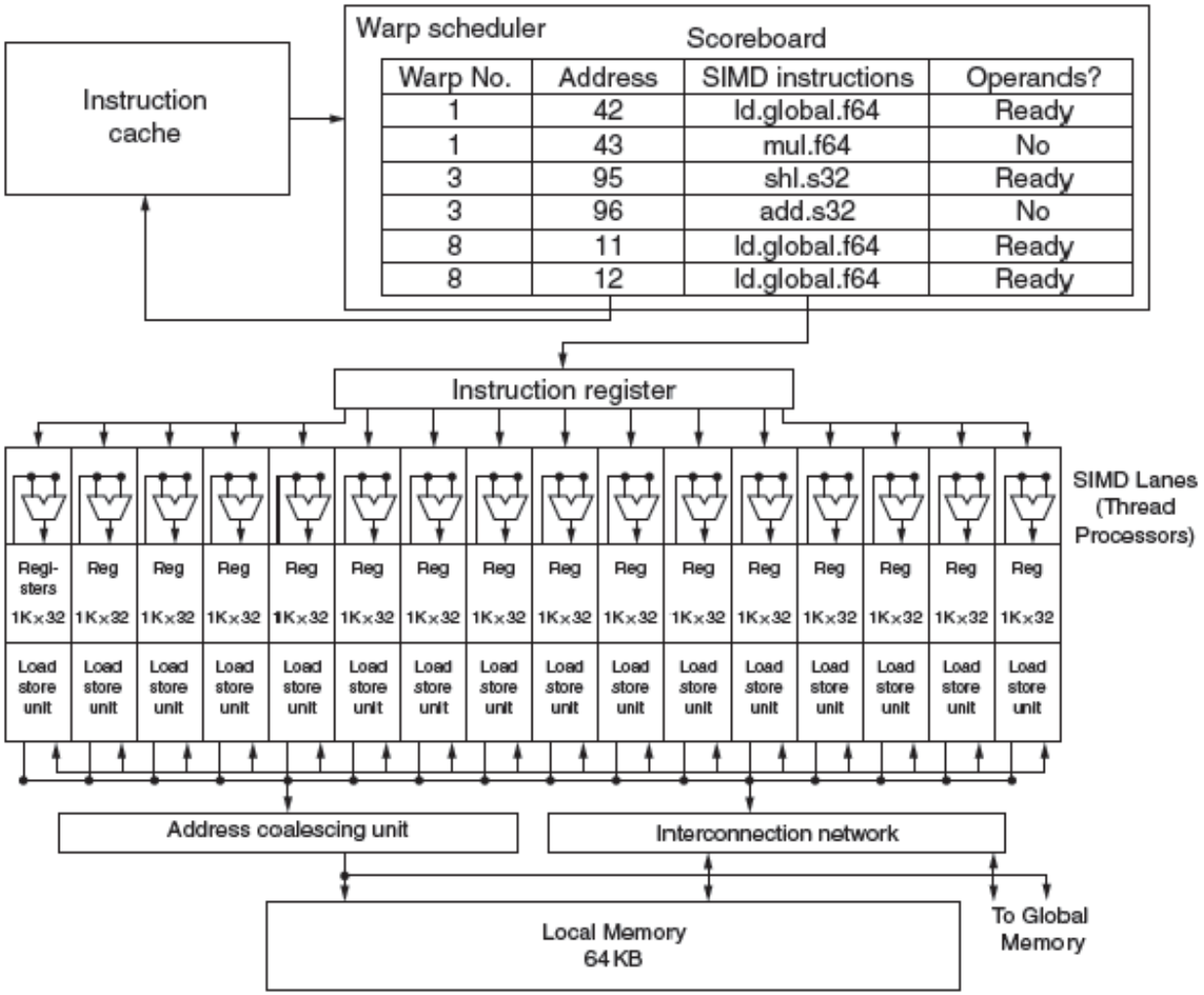
Example: X * Y (8,192 elements)

- A *Grid* works on the whole 8,192 elements
- A grid is composed of *Thread Blocks*, each processing 512 elements
  - # of blocks = 8,192/512 = 16
- A SIMD instruction executes 32 elements at a time
  - # of SIMD threads in a block = 512/32 = 16

Programmer's View of Execution

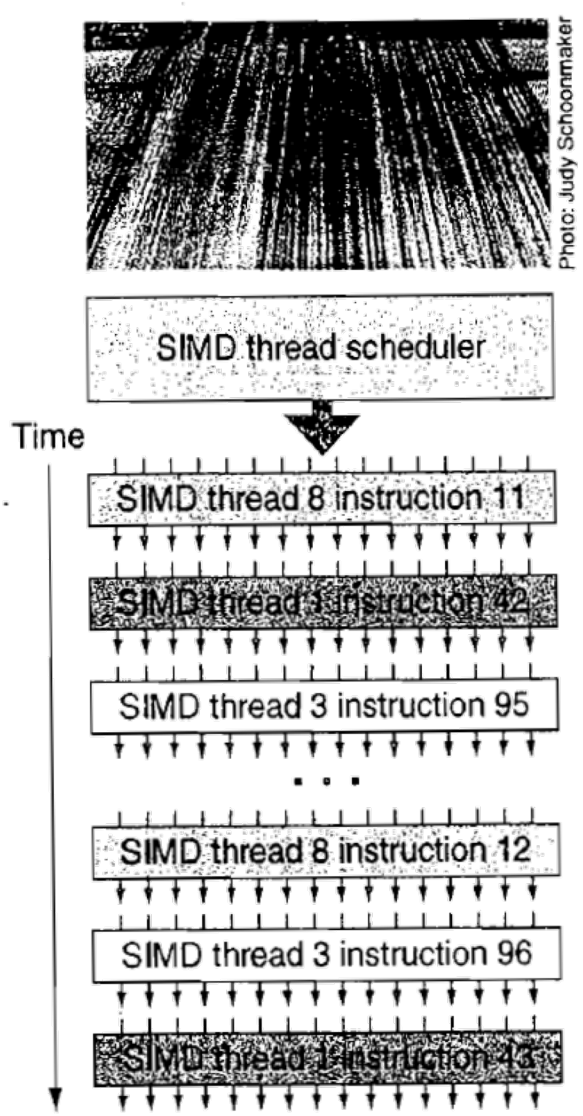


Multithreaded SIMD Processor

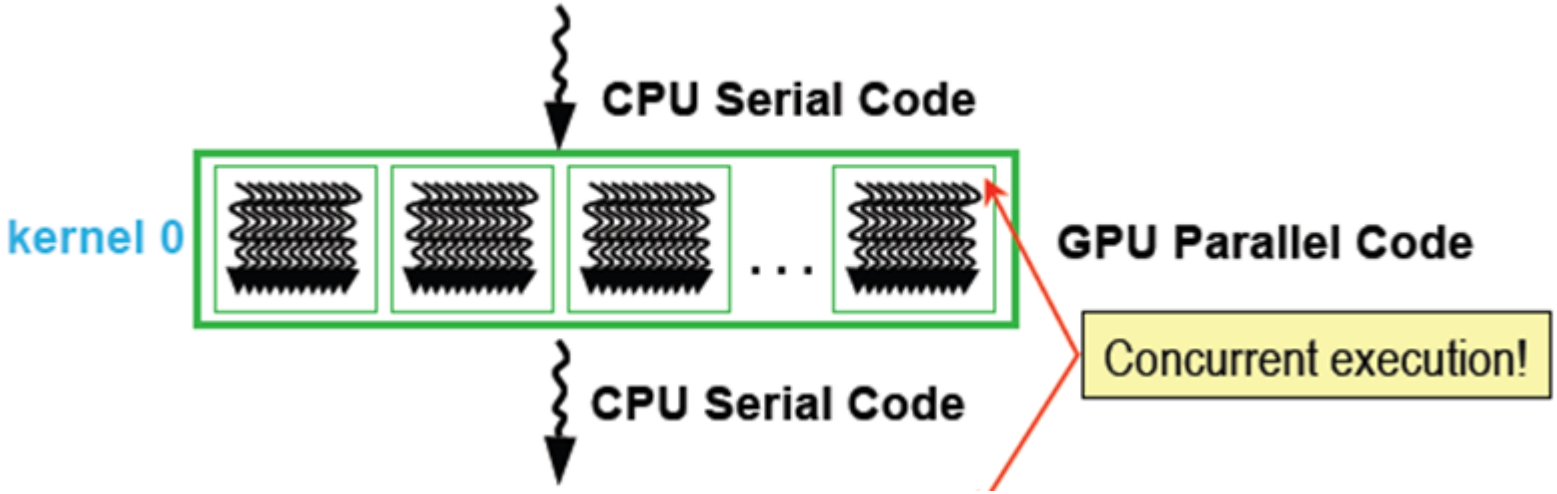


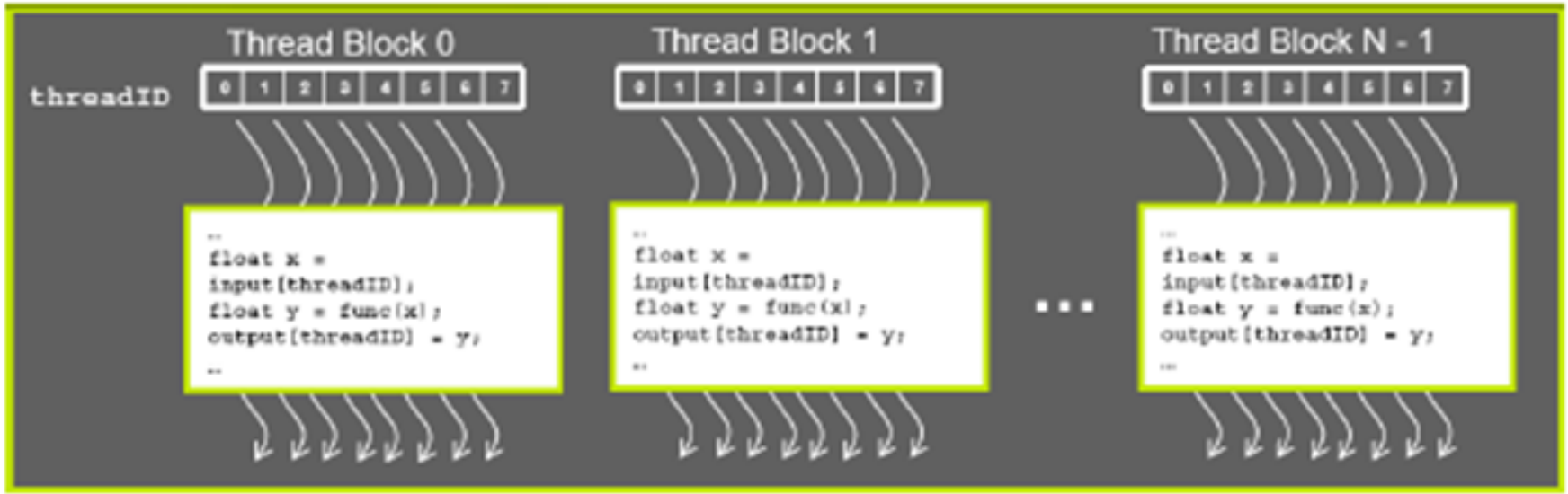
Scheduling of Threads of SIMD Instructions

- Each multithreaded SIMD processor has *one scheduler*
- The scheduler selects a *ready thread of SIMD instructions* and issues an instruction synchronously to *all the SIMD lanes* executing the SIMD thread
- Threads of SIMD instructions are *independent*, so the scheduler may select a different SIMD thread each time

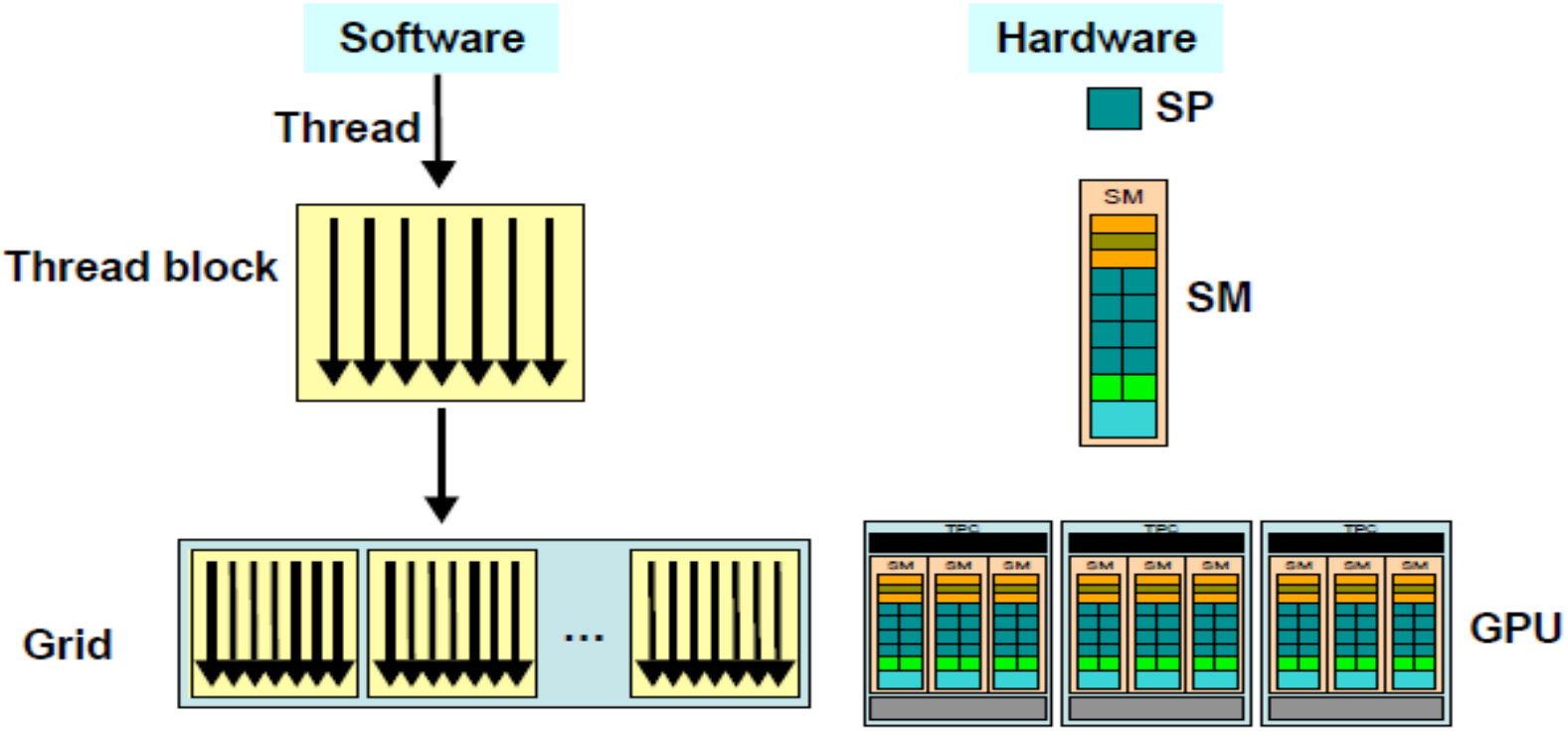


CUDA Threads





Parallel Program Organization in CUDA



CUDA Program Example

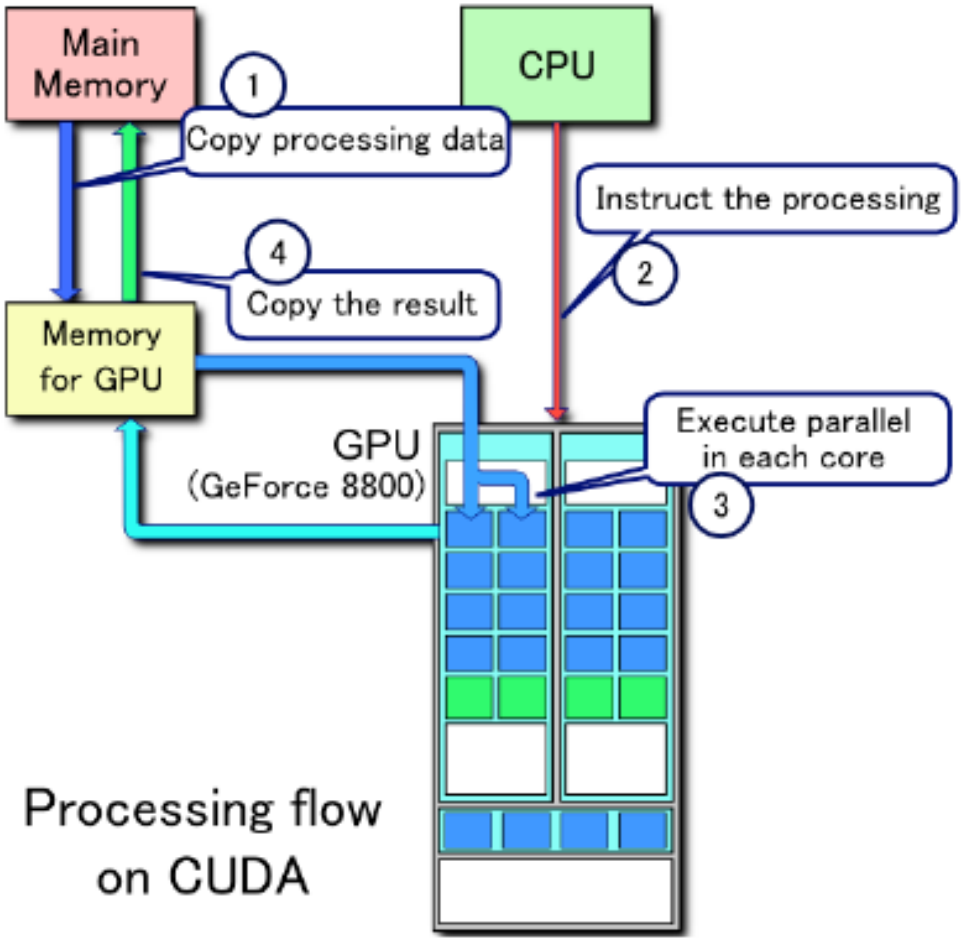
```
//CPU program
//sum of two vectors a and b
void add_cpu(float *a, float *b, int N)
{
    for (int idx = 0; idx<N; idx++)
        a[idx] += b[idx];
}


void main()
{
    .....
    fun_add(a, b, N);
}
```

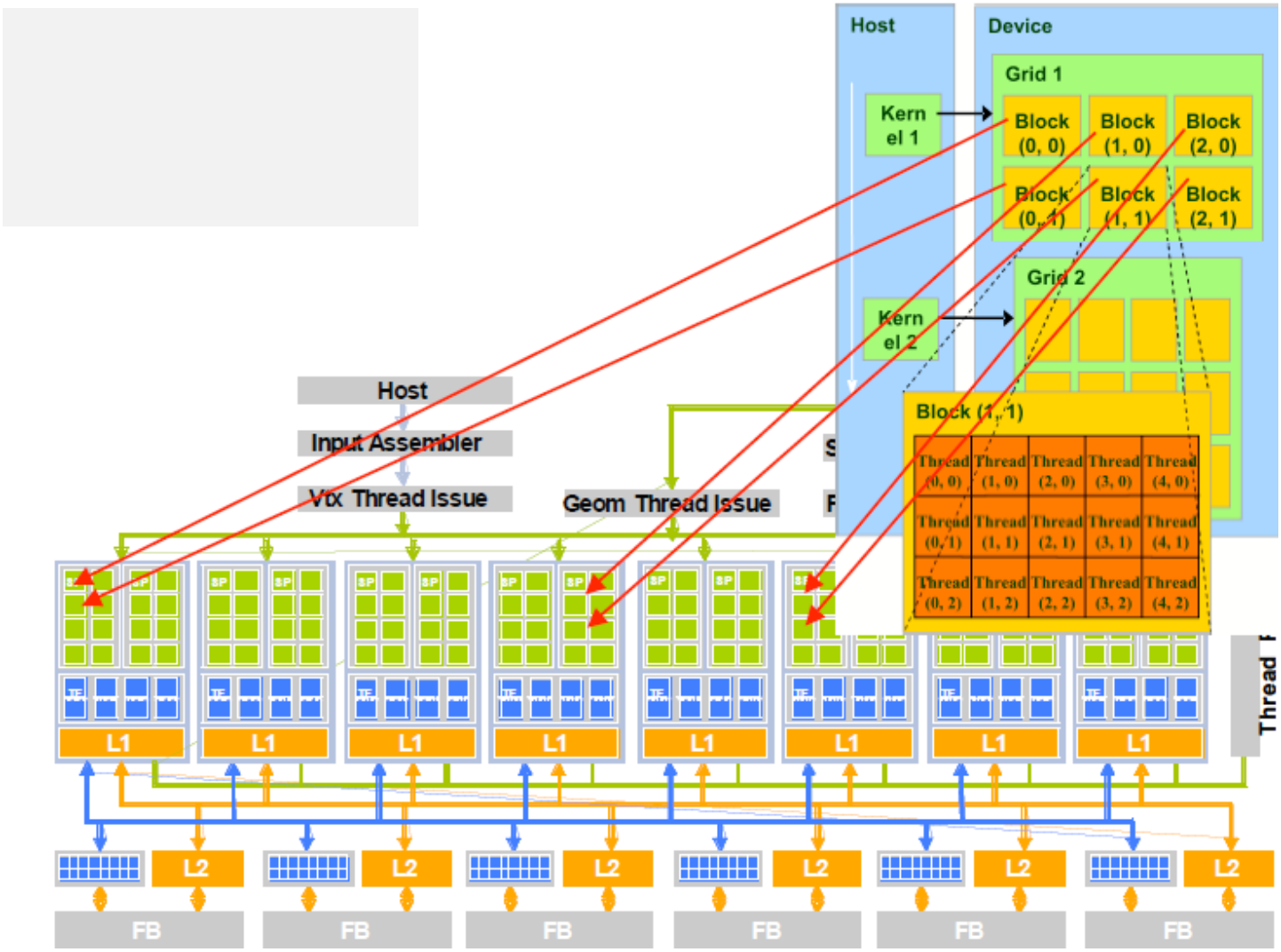
```
//CUDA program
//sum of two vectors a and b
__global__ void add_gpu(float *a, float *b, int N)
{
    Int idx =blockIdx.x* blockDim.x+ threadIdx.x;
    if (idx < N)
        a[idx] += b[idx];
}

void main()
{
    .....
    dim3 dimBlock (256);
    dim3 dimGrid( ceil( N / 256 );
    fun_add<<<dimGrid, dimBlock>>>(a, b, N);
}
```

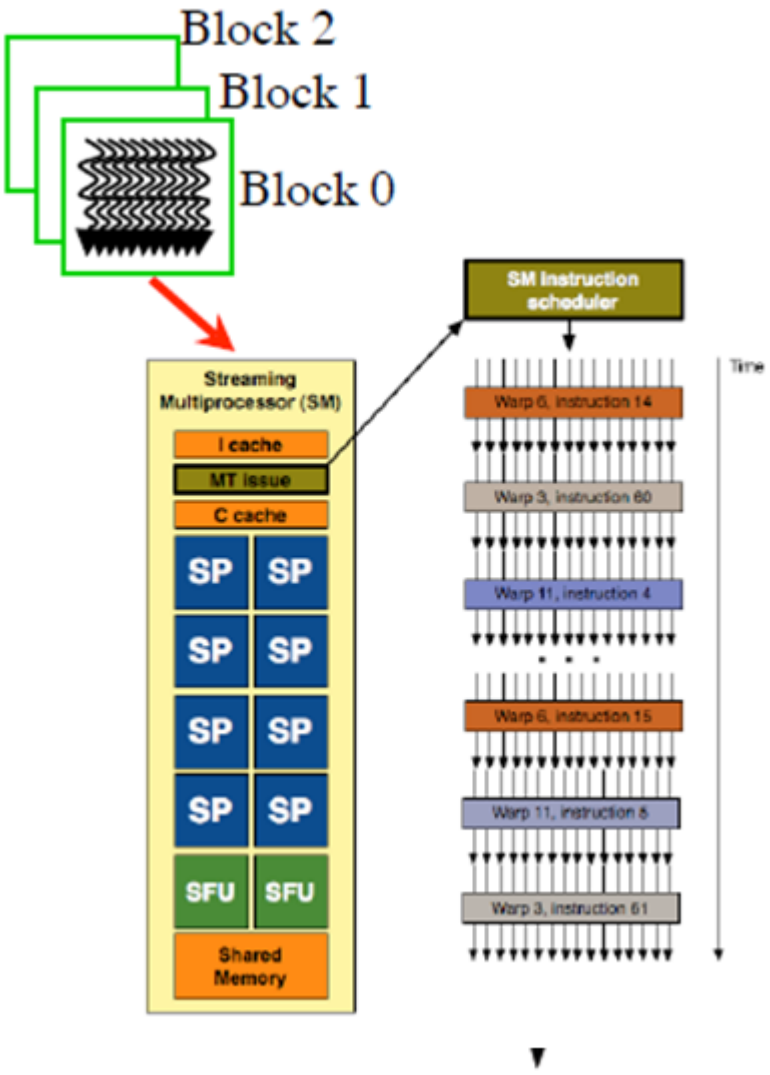
CUDA Processing Flow



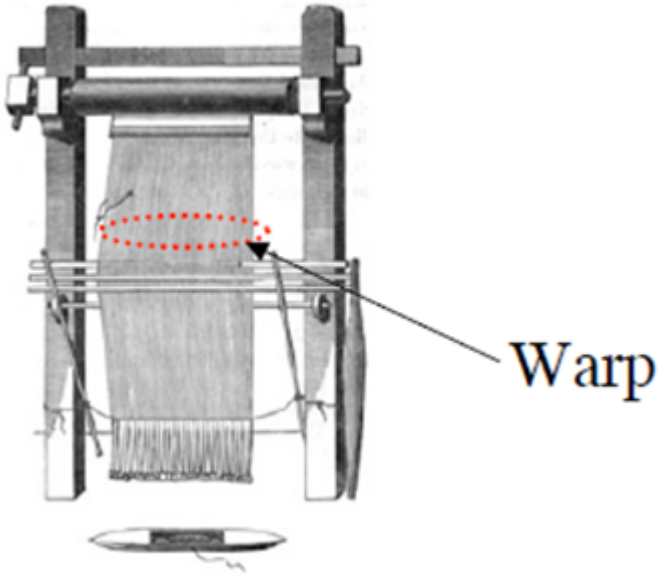
Mapping Threads to GPU



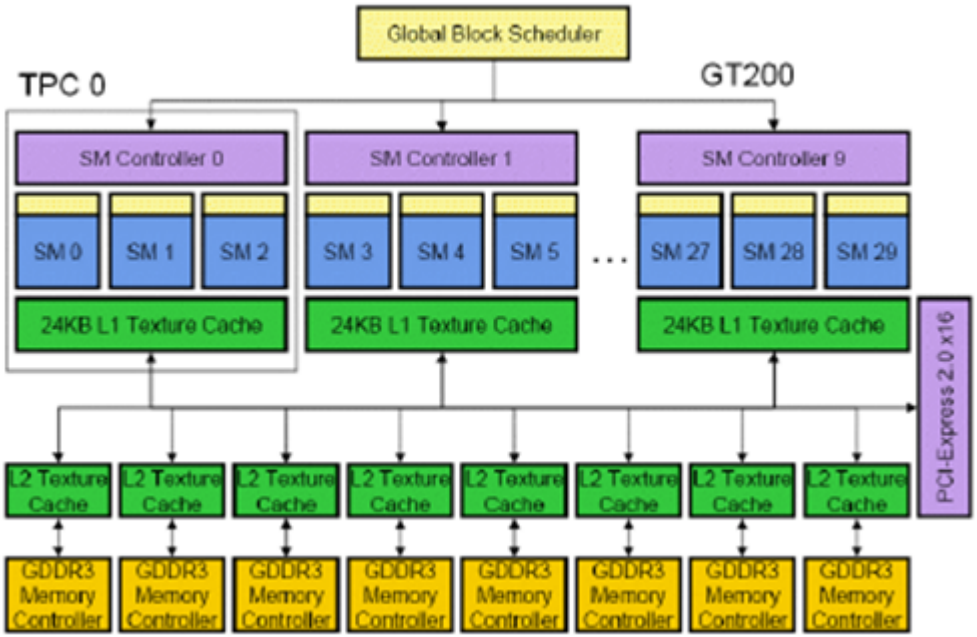
Block of Threads vs. SM



- A Warp: Threads execute the same instructions at same time
- A SP runs one CUDA thread



Global Block Scheduler



- The *global block scheduler* manages and allocates blocks of threads to SM
- Load balancing

Life Cycle of Thread

- Grid is started on GPU
- A block of threads allocated to a SM
- SM organizes threads of a given block into warps
- Warps are scheduled on SM