Chapter 4

Data-Level Parallelism in Vector, SIMD, and GPU Architectures

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 per data operation
  - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially

SIMD Variations

- Vector architectures
- SIMD extensions
- Graphics Processor Units (GPUs)

Variation 1: Vector Architectures

- They are easier to understand and to compile to than other two variations
- But they are considered too expensive for microprocessors until recently
  - —Part of expense was in transistors
  - —Part was in the cost of sufficient DRAM bandwidth
  - —Given the widespread reliance on caches

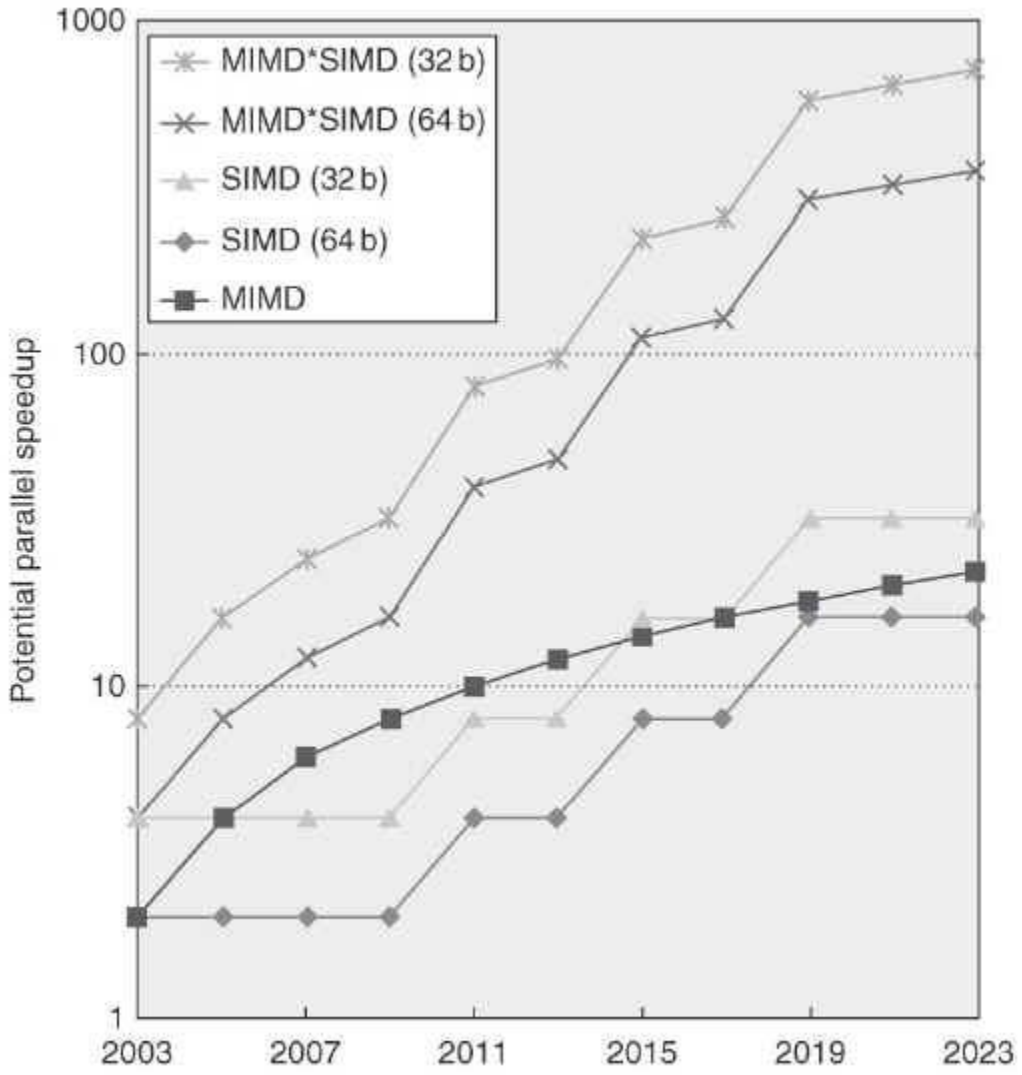
Variation 2: SIMD Extensions

- They mean parallel execution of data operations
- They can be found in most ISAs to support multimedia applications
- For x86 architectures
  - Multimedia Extensions (MMX)
  - Streaming SIMD Extensions (SSE)
  - Advanced Vector Extensions (AVE)
- To get the highest computation rate from an x86 computer, you often need to use these SIMD instructions, especially for floating-point programs

Variation 3: Graphics Processing Units (GPUs)

- They offer higher potential performance than is found in traditional multicore computers today
- This environment has a system processor and system memory in addition to the GPU and its graphics memory
- The GPU community refers to this type of architecture as heterogeneous

Potential Speedup via Parallelism from MIMD, SIMD



SIMD Parallelism (1)

- For problems with lots of data parallelism, all three SIMD variations share the advantage of being easier for the programmer than classic parallel MIMD programming
- For x86 processors:
  - Expect two additional cores per chip per year
  - SIMD width to double every four years
  - Potential speedup from SIMD to be twice that from MIMD!
- It's as least as important to understand SIMD parallelism as MIMD parallelism

SIMD Parallelism (2)

- The goal is for architects to understand why vector is more general than multimedia MIMD, as well as the similarities and differences between vector and GPU architectures
- Since vector architectures are supersets of the multimedia SIMD instructions, including a better model for compilation
- GPUs share several similarities with vector architectures

Vector Architectures

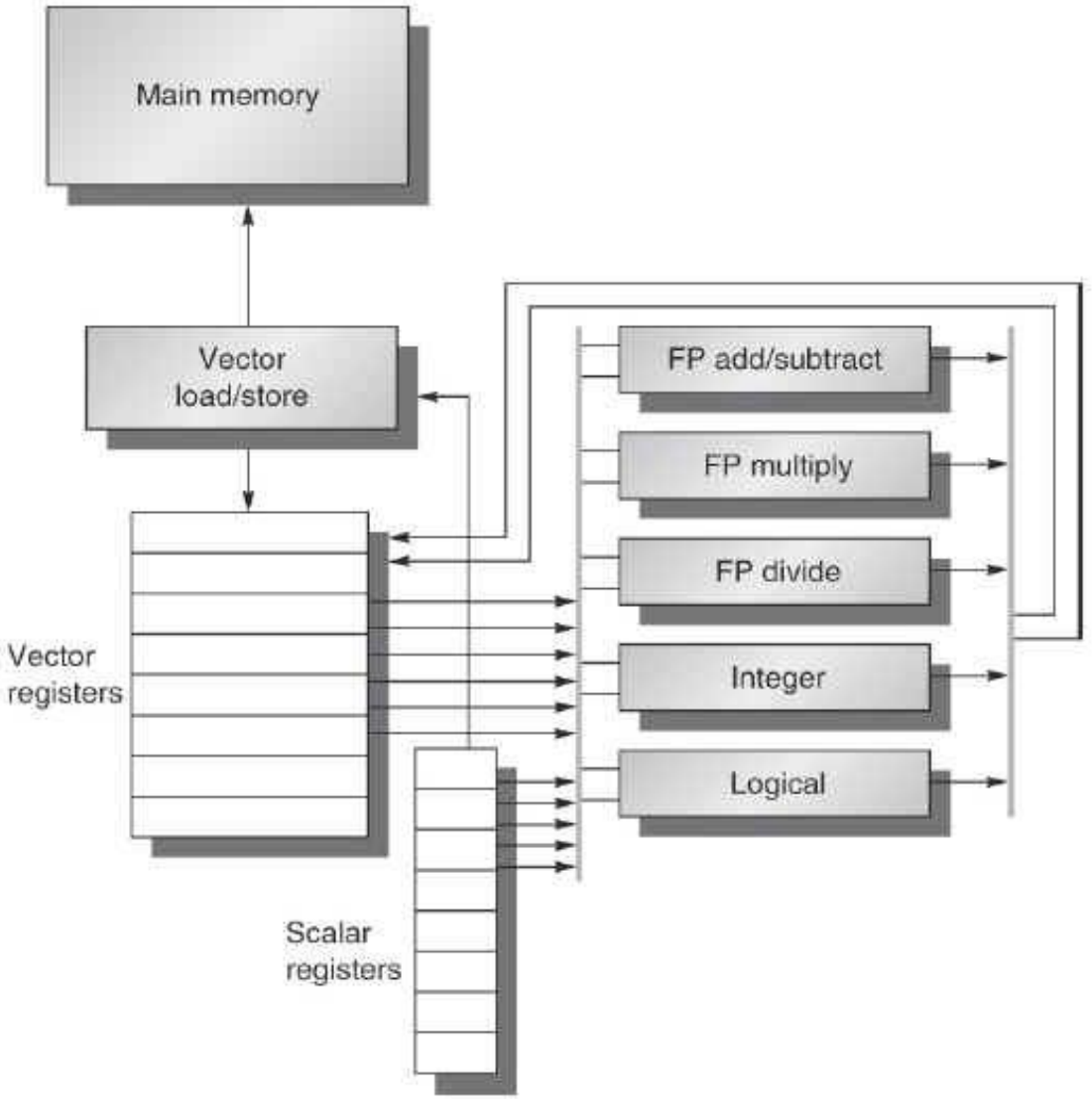
- Basic idea:
  - —Read sets of data elements into “vector registers”
  - —Operate on those registers
  - —Disperse the results back into memory

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

VMIPS

- Example architecture: VMIPS
  - —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

VMIPS Architecture



VMIPS Instructions

- ADDVV.D: add two vectors
- ADDVS.D: add vector to a scalar
- LV/SV: vector load and vector store from address

- Example: DAXPY

```
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     V4,V2,V3      ; add
SV        Ry,V4         ; store the result
```

- Requires 6 instructions vs. almost 600 for MIPS

VMIPS 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.

How Vector Processors Work: Example

- Take a typical vector problem from Linpack: *SAXPY* or *DAXPY*

$$Y = a * X + Y$$

- Show the code for MIPS and VMIPS for the DAXPY loop. Assume that the starting address of X and Y are in Rx and Ry, respectively.

Answer: How Vector Processors Work

```
        LD      F0,     a               ; load scalar a
        ADDI    R4,     Rx,     #512    ; last address to load
Loop:   LD      F2,     0(Rx)           ; load X(i)
        MULTD   F2,     F0,     F2      ; a x X(i)
        LD      F4,     0 (Ry)          ; load Y(i)
        ADDD    F4,     F2,     F4      ; a x X(i) + Y(i)
        SD      F4,     0 (Ry)          ; store into Y(i)
        ADDI    Rx,     Rx, #8          ; increment index to X
        ADDI    Ry,     Ry, #8          ; increment index to Y
        SUB     R20,    R4,     Rx      ; compute bound
        BNZ     R20,    loop            ; check if done
```

VMIPS code for DAXPY

```
        LD      F0,     a               ; load scalar a
        LV      V1,     Rx              ; load vector X
        MULTSV  V2,     F0,     V1      ; vector-scalar multiply
        LV      V3,     Ry               ; load vector Y
        ADDV    V4,     V2,     V3      ; add
        SV      Ry,     V4              ; store the result
```

Vector Execution Time

- Execution time 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

- *Convoy*
  - —Set of vector instructions that could potentially execute together

Chimes

- Sequences with read-after-write dependency hazards can be in the same convey via *chaining*

- *Chaining*
  - —Allows a vector operation to start as soon as the individual elements of its vector source operand become available

- *Chime*
  - —Unit of time to execute one convey
  - —*m* conveys executes in *m* chimes
  - —For vector length of *n*, requires *m* x *n* clock cycles

Example

```
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 two vectors
SV        Ry,V4      ;store the sum
```

Convoys:

1	LV	MULVS.D
2	LV	ADDVV.D
3	SV	

3 chimes, 2 FP ops per result, cycles per FLOP = 1.5

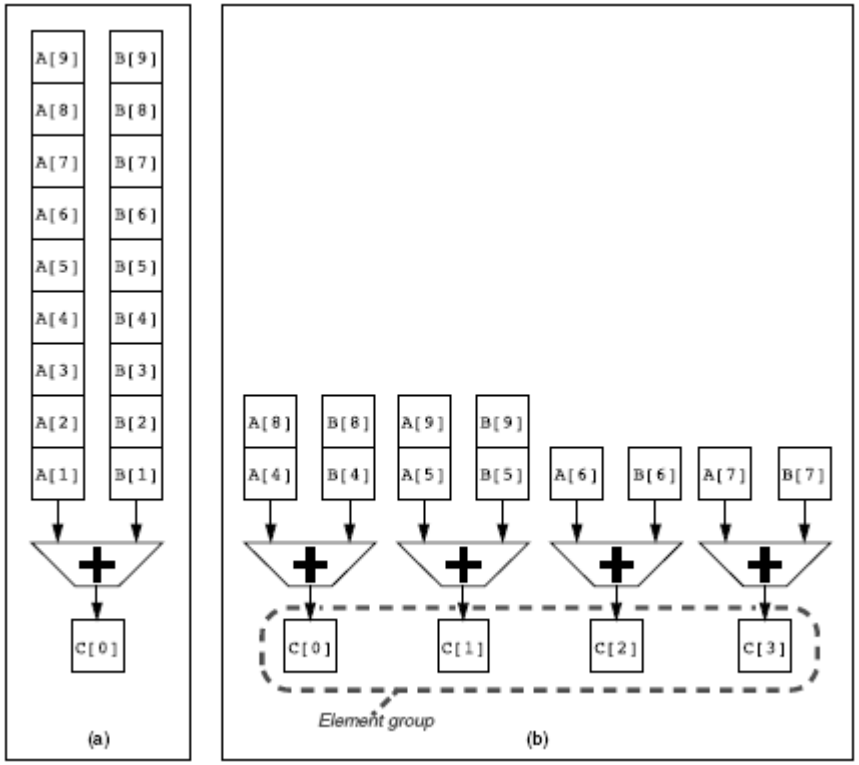
For 64 element vectors, requires 64 x 3 = 192 clock cycles

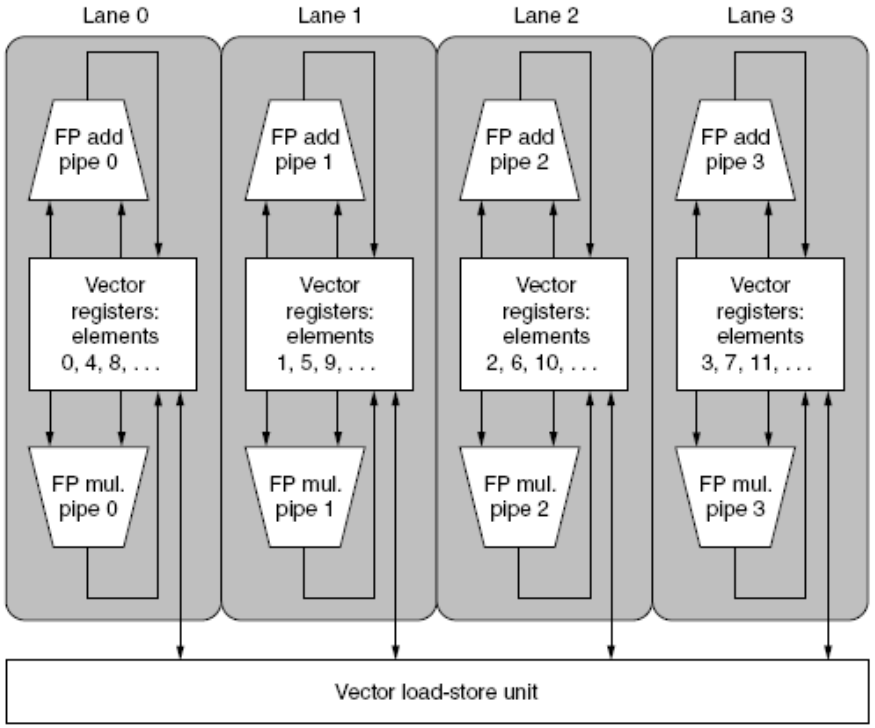
Challenges

- Start up time
  - Latency of vector functional unit
  - Assume the same as Cray-1
    - Floating-point add => 6 clock cycles
    - Floating-point multiply => 7 clock cycles
    - Floating-point divide => 20 clock cycles
    - Vector load => 12 clock cycles
- Improvements:
  - > 1 element per clock cycle
  - Non-64 wide vectors
  - IF statements in vector code
  - Memory system optimizations to support vector processors
  - Multiple dimensional matrices
  - Sparse matrices
  - Programming a vector computer

Multiple Lanes

- Element *n* of vector register *A* is "hardwired" to element *n* of vector register *B*
  - Allows for multiple hardware lanes

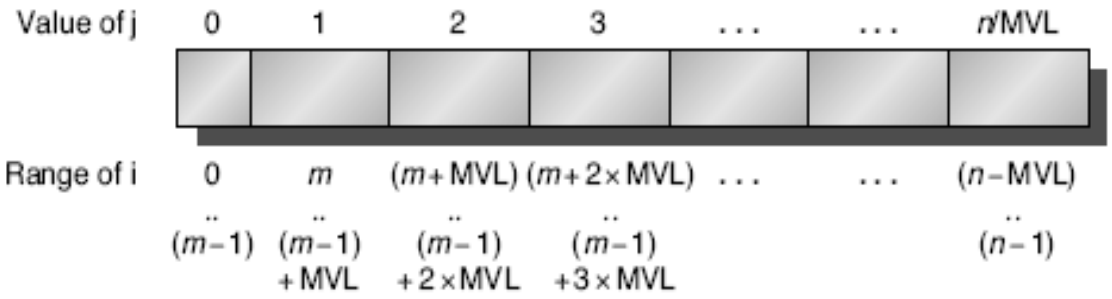




Vector Length Register

- Vector length not known at compile time?
- Use Vector Length Register (VLR)
- Use strip mining for vectors over the maximum length:

```
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
    for (i = low; i < (low+VL); i=i+1) /*runs for length VL*/
       Y[i] = a * X[i] + Y[i] ; /*main operation*/
    low = low + VL; /*start of next vector*/
    VL = MVL; /*reset the length to maximum vector length*/
}
```



Vector Mask Registers

- Consider:

```
for (i = 0; i < 64; i=i+1)
        if (X[i] != 0)
                X[i] = X[i] – Y[i];
```

- Use vector mask register to "disable" elements:

```
LV          V1,Rx       ;load vector X into V1
LV          V2,Ry       ;load vector Y
L.D         F0,#0       ;load FP zero into F0
SNEVS.D     V1,F0       ;sets VM(i) to 1 if V1(i)!=F0
SUBVV.D     V1,V1,V2    ;subtract under vector mask
SV          Rx,V1       ;store the result in X
```

- GFLOPS rate decreases!

Memory Banks

- Memory system must be designed to support high bandwidth for vector loads and stores
- Spread accesses across multiple banks
  - Control bank addresses independently
  - Load or store non sequential words
  - Support multiple vector processors sharing the same memory

- Example:
  - 32 processors, each generating 4 loads and 2 stores/cycle
  - Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns
  - How many memory banks needed?

Example

- The largest configuration of a Cray 70 (Gray T7932) has 32 processors, each capable of generating 4 loads and 2 stores per cycle. The processor cycle time is 2.167 ns, while SRAM cycle time is 15 ns. How many memory banks needed?
- Answer:
  - Maximum memory reference/cycle = 6 x 32 = 192
  - SRAM is busy 15/2.167 = 6.92 $\cong$ 7 cycles
  - Thus, a minimum memory banks is 192 x 7 = 1344

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*
- Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:
  - #banks / LCM(stride,#banks) < bank busy time

Example

- Suppose we have 8 memory banks with a bank busy time of 6 clocks and a total memory latency of 12 cycles. How long will it take to complete a 64-element vector load with a stride of 1? With a stride of 32?
- Answer:
  - since # of banks > the bank busy time, for a stride of the load will take 12 + 64 = 76 clock cycles,
  - or 1.2 clock cycles per element
  - the worst possible stride is a value that is a multiple of the number of memory banks
  - as in the case with a stride of 32 and 8 memory banks
  - every access to memory (after the first one) will collide with the previous access and will have to wait for the 6-clock-cycle bank busy time
  - the total time will be 12 + 1 + 6 x 63 = 391 cycles, or 6.1 cycles per element

Scatter-Gather

- Consider:

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

- Use index vector:

```
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[]]
```

Programming Vec. 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

SIMD Extensions

- Media applications operate on data types narrower than the native word size
  - Example: disconnect carry chains to "partition" adder

- Limitations, compared to vector instructions:
  - Number of data operands encoded into op code
  - No sophisticated addressing modes (strided, scatter-gather)
  - No mask registers

Typical Multimedia SIMD Instructions

Instruction category	Operands
Unsigned add/subtract	Thirty-two 8-bit, sixteen 16-bit, eight 8-bit, or four 64-bit
Maximun/minimum	Thirty-two 8-bit, sixteen 16-bit, eight 8-bit, or four 64-bit
Average	Thirty-two 8-bit, sixteen 16-bit, eight 8-bit, or four 64-bit
Shift right/left	Thirty-two 8-bit, sixteen 16-bit, eight 8-bit, or four 64-bit
Floating point	Sixteen 16-bit, eight 8-bit, four 64-bit, or two 128-bit

SIMD Implementations

- Implementations:
  - Intel MMX (1996)
    - Eight 8-bit integer ops or four 16-bit integer ops
  - Streaming SIMD Extensions (SSE) (1999)
    - Eight 16-bit integer ops
    - Four 32-bit integer/fp ops or two 64-bit integer/fp ops
  - Advanced Vector Extensions (2010)
    - Four 64-bit integer/fp ops
  - Operands must be consecutive and aligned memory locations

AVX Instructions for x86 Architectures

Instruction	Description
VADDPD	Add four packed double-precision operands
VSUBPD	Subtract four packed double-precision operands
VMULPD	Multiply four packed double-precision operands
VDIVPD	Divide four packed double-precision operands
VFMADDPD	Multiply and add four packed double-precision operands
VFMSUBPD	Multiply and subtract four packed double-precision operands
VCMPxx	Compare four packed double-precision operands for EQ, NEQ, LT, LE, GT, GE, …
VMOVAPD	Move aligned four packed double-precision operands
VBROADCASTSD	Broadcast one packed double-precision operands to four locations in a 256-bit register

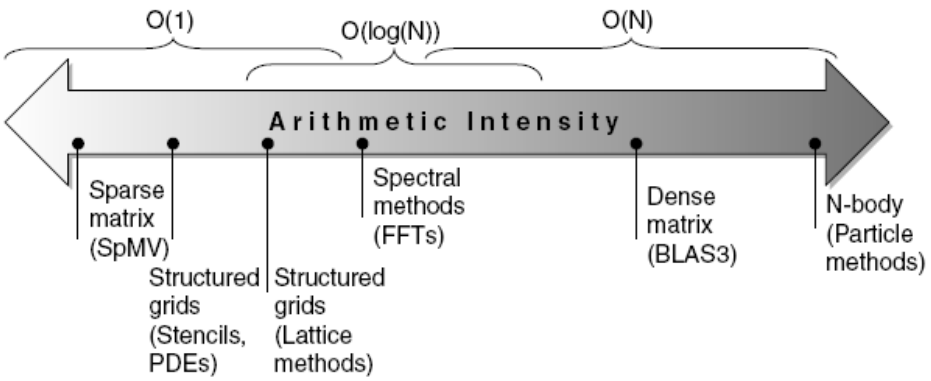
Example SIMD Code

- Example DXPY:

```
        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
```

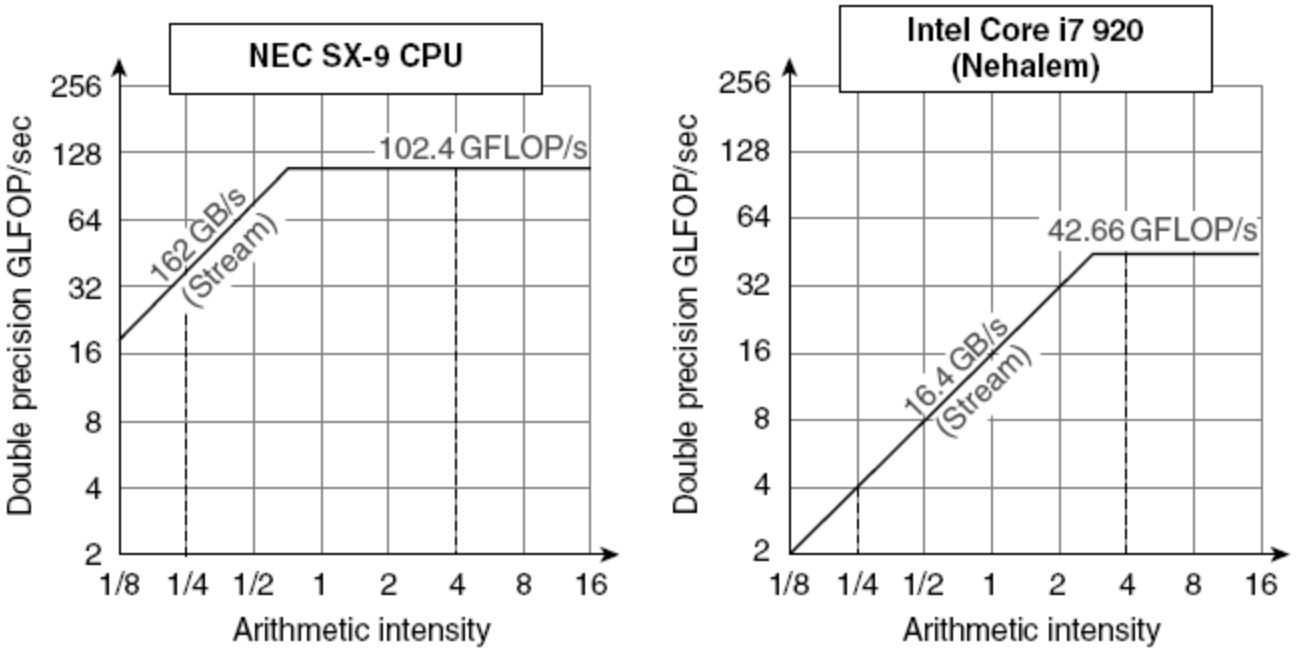
Roofline Performance Model

- Basic idea:
  - —Plot peak floating-point throughput as a function of arithmetic intensity
  - —Ties together floating-point performance and memory performance for a target machine
- Arithmetic intensity
  - —Floating-point operations per byte read



Examples

- Attainable GFLOPs/sec Min = (Peak Memory BW × Arithmetic Intensity, Peak Floating Point Perf.)

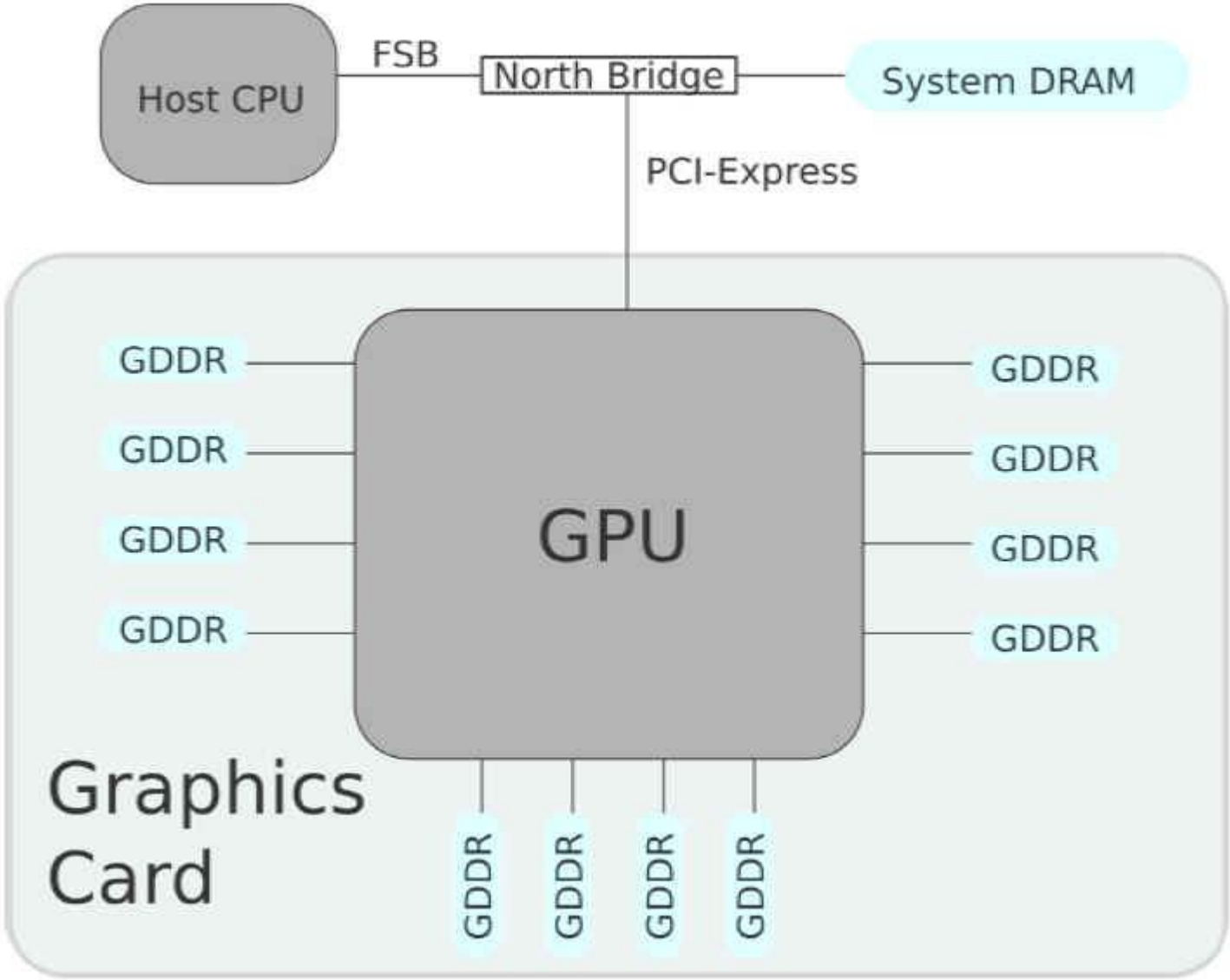


Graphical Processing Units

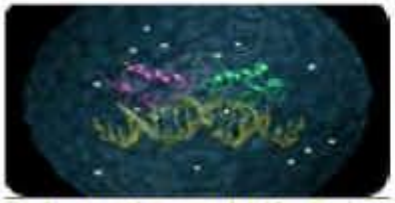
- Given the hardware invested to do graphics well, how can be supplement it to improve performance of a wider range of applications?

- 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"

System Architecture

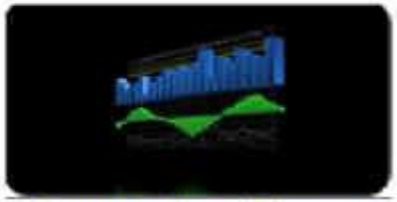


General Purposed Computing



Bio-Informatics and Life Sciences

Computational Electromagnetics and Electrodynamics



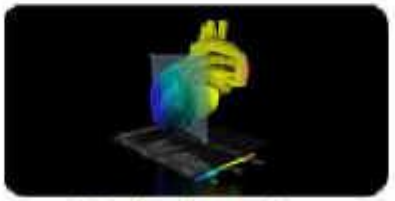
Computational Finance

Computational Fluid Dynamics

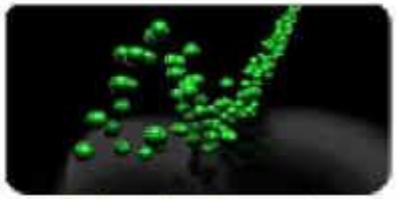
Data Mining, Analytics, and Databases

Imaging and Computer Vision

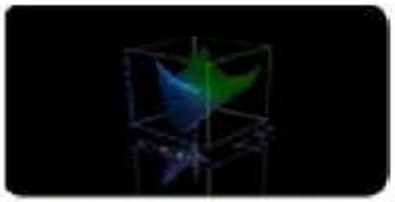
MATLAB Acceleration



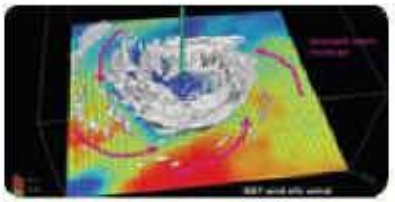
Medical Imaging



Molecular Dynamics

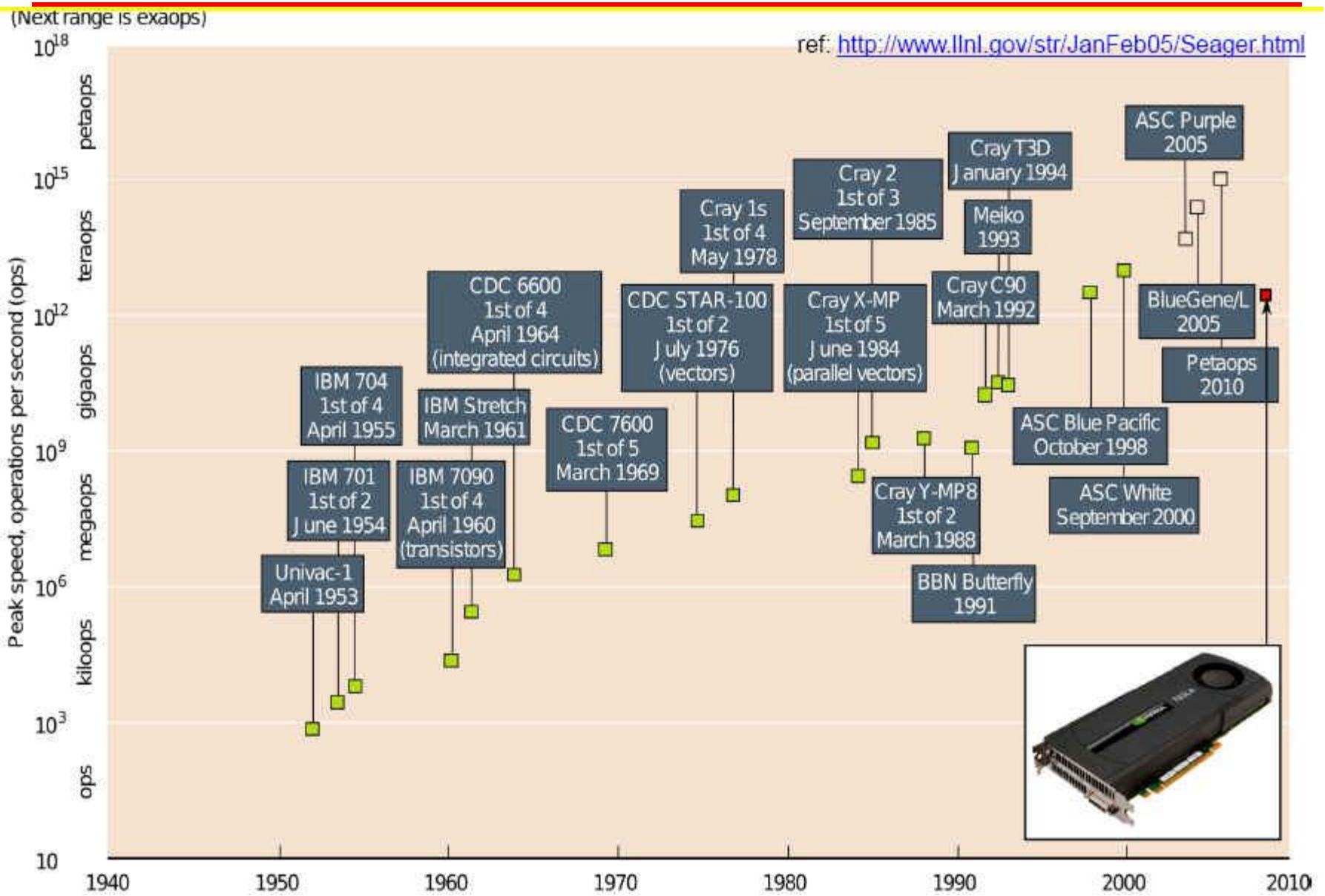


Numerical Packages



Weather, Atmospheric, Ocean Modeling, and Space Sciences

Single-Chip GPU vs. Fastest Super Computers



Top500 Super Computer in June 2010

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron Six Core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00	6950.60
2	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU / 2010 Dawning	120640	1271.00	2984.30	
3	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband / 2009 IBM	122400	1042.00	1375.78	2345.50
4	National Institute for Computational Sciences/University of Tennessee United States	Kraken XT5 - Cray XT5-HE Opteron Six Core 2.6 GHz / 2009 Cray Inc.	98928	831.70	1028.85	
5	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution / 2009 IBM	294912	825.50	1002.70	2268.00
6	NASA/Ames Research Center/NAS United States	Pleiades - SGI Altix ICE 8200EX/8400EX, Xeon HT QC 3.0/Xeon Westmere 2.93 Ghz, Infiniband / 2010 SGI	81920	772.70	973.29	3096.00
7	National SuperComputer Center in Tianjin/NUDT China	Tianhe-1 - NUDT TH-1 Cluster, Xeon E5540/E5450, ATI Radeon HD 4870 2, Infiniband / 2009 NUDT	71680	563.10	1206.19	
8	DOE/NNSA/LLNL United States	BlueGene/L - eServer Blue Gene Solution / 2007 IBM	212992	478.20	596.38	2329.60
9	Argonne National Laboratory United States	Intrepid - Blue Gene/P Solution / 2007 IBM	163840	458.61	557.06	1260.00
10	Sandia National Laboratories / National Renewable Energy Laboratory United States	Red Sky - Sun Blade x6275, Xeon X55xx 2.93 Ghz, Infiniband / 2010 Sun Microsystems	42440	433.50	497.40	

NVIDIA Tesla C2050

ATI Radeon HD 4870 2

GPU Will Top the List in Nov 2010

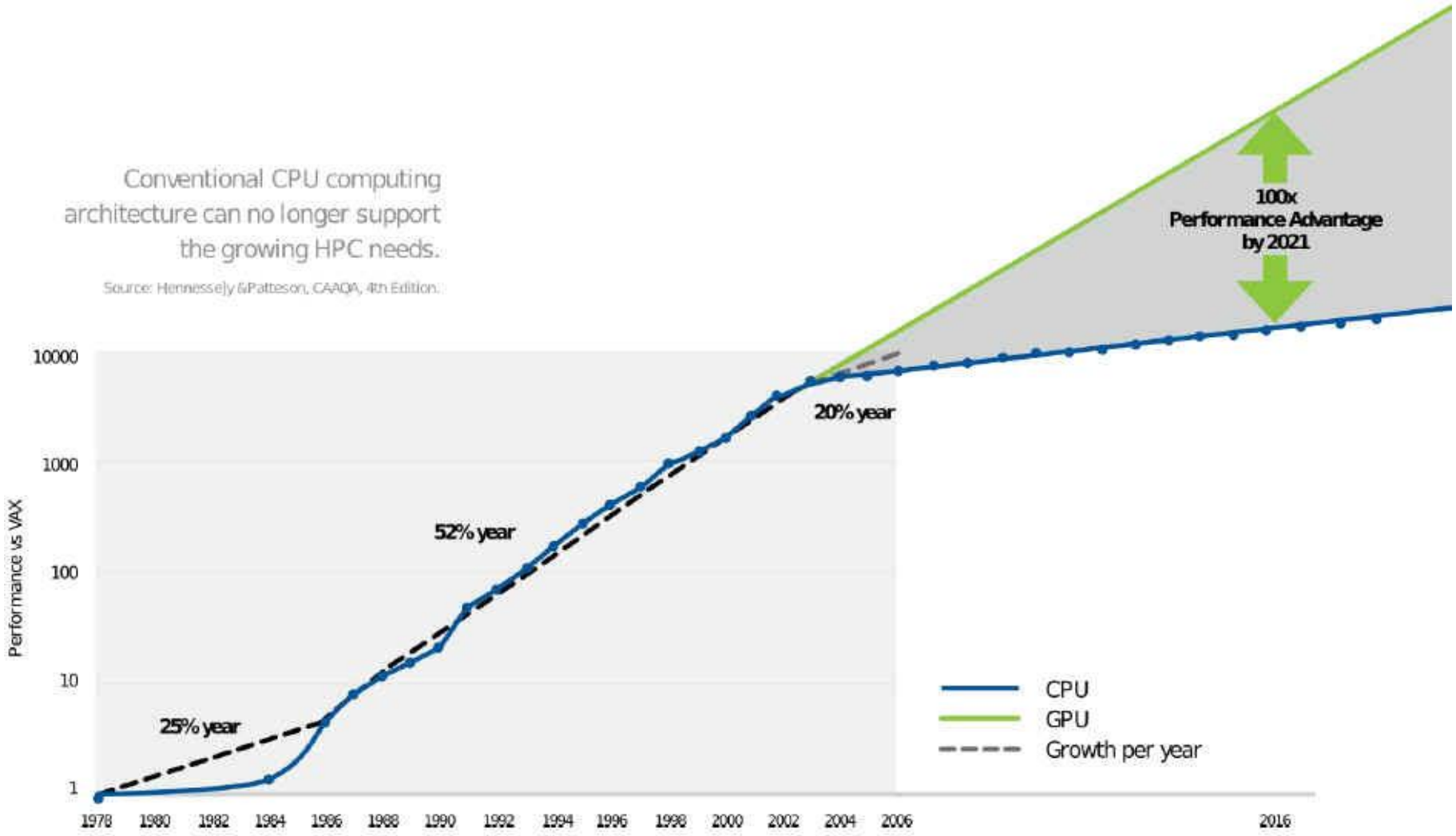
NVIDIA Tesla M2050

NVIDIA Tesla C2050

ATI Radeon HD 4870 2

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
To-be #1	National SuperComputing Center in Tianjin/NUDT, China	Tianhe-1A - Xeon NVidia Tesla M2050	14336 CPU 7168 GPU	2500		
1	Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron Six Core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00	6950.60
2	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU / 2010 Dawning	120640	1271.00	2984.30	
3	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband / 2009 IBM	122400	1042.00	1375.78	2345.50
4	National Institute for Computational Sciences/University of Tennessee United States	Kraken XT5 - Cray XT5-HE Opteron Six Core 2.6 GHz / 2009 Cray Inc.	98928	831.70	1028.85	
5	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution / 2009 IBM	294912	825.50	1002.70	2268.00
6	NASA/Ames Research Center/NAS United States	Pleiades - SGI Altix ICE 8200EX/8400EX, Xeon HT QC 3.0/Xeon Westmere 2.93 Ghz, Infiniband / 2010 SGI	81920	772.70	973.29	3096.00
7	National SuperComputer Center in Tianjin/NUDT China	Tianhe-1 - NUDT TH-1 Cluster, Xeon E5540/E5450, ATI Radeon HD 4870 2, Infiniband / 2009 NUDT	71680	563.10	1206.19	
8	DOE/NNSA/LLNL United States	BlueGene/L - eServer Blue Gene Solution / 2007 IBM	212992	478.20	596.38	2329.60
9	Argonne National Laboratory United States	Intrepid - Blue Gene/P Solution / 2007 IBM	163840	458.61	557.06	1260.00
10	Sandia National Laboratories / National Renewable Energy Laboratory United States	Red Sky - Sun Blade x6275, Xeon X55xx 2.93 Ghz, Infiniband / 2010 Sun Microsystems	42440	433.50	497.40	

The Gap Between CPU and GPU



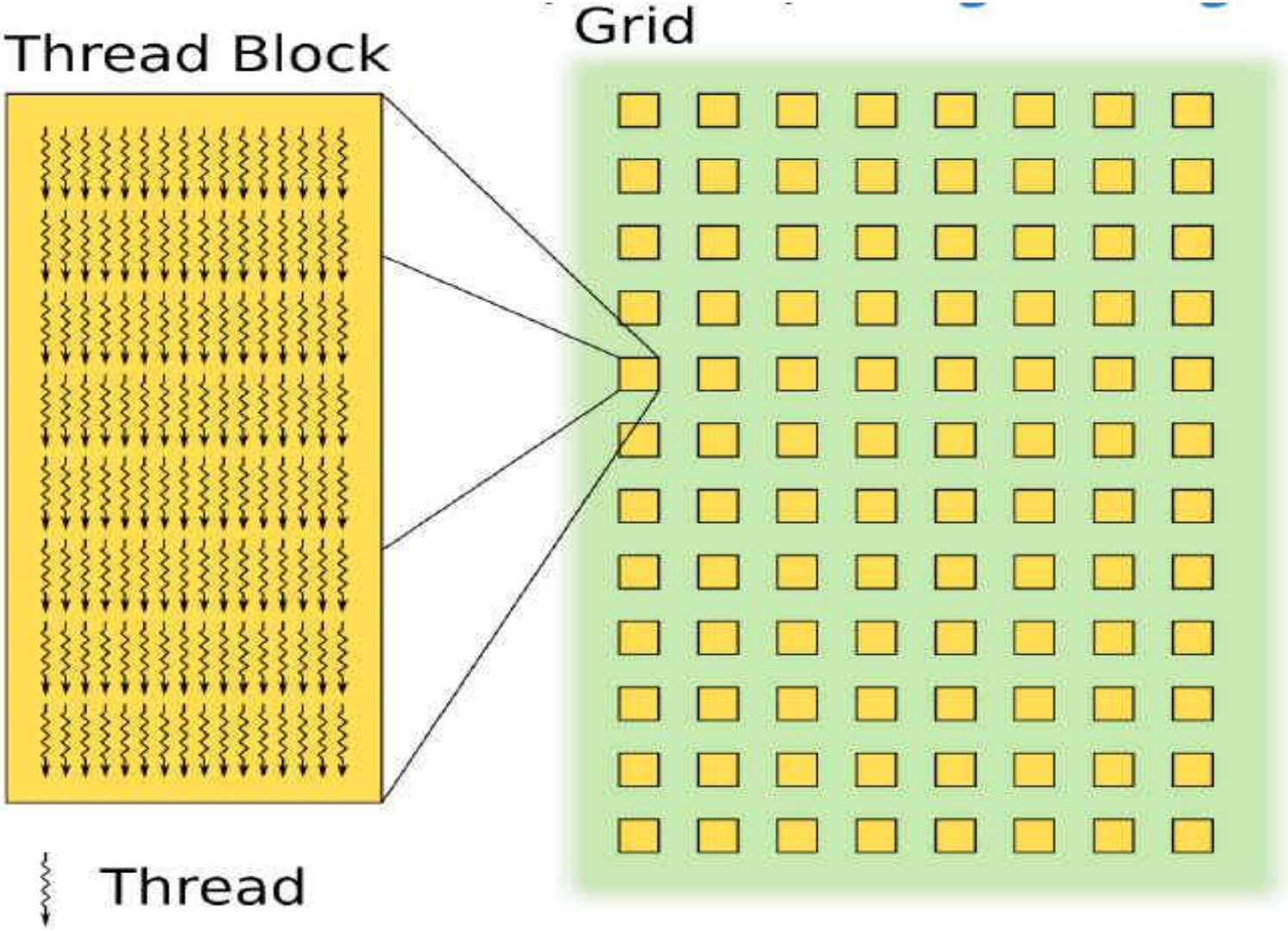
Threads and Blocks

- 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

- Massive number (> 10000) of light-weight threads.



Multicore with Multimedia SIMD Extensions vs. Recent GPUs

Feature	Multicore with SIMD	GPU
SIMD processors	4 to 8	8 to 16
SIMD lanes/processor	2 to 4	8 to 16
Multithreading hardware support for SIMD threads	2 to 4	16 to 32
Typical ratio of single-precision to double-precision performance	2:1	2:1
Largest cache size	8 MB	0.75 MB
Size of memory address	64-bit	64-bit
Size of main memory	8 GB to 256 GB	4 to 6 GB
Memory protection at level of page	Yes	Yes
Demand paging	Yes	No
Integrated scalar processor/SIMD processor	Yes	No
Cache coherent	Yes	No

NVIDIA GPU Architecture

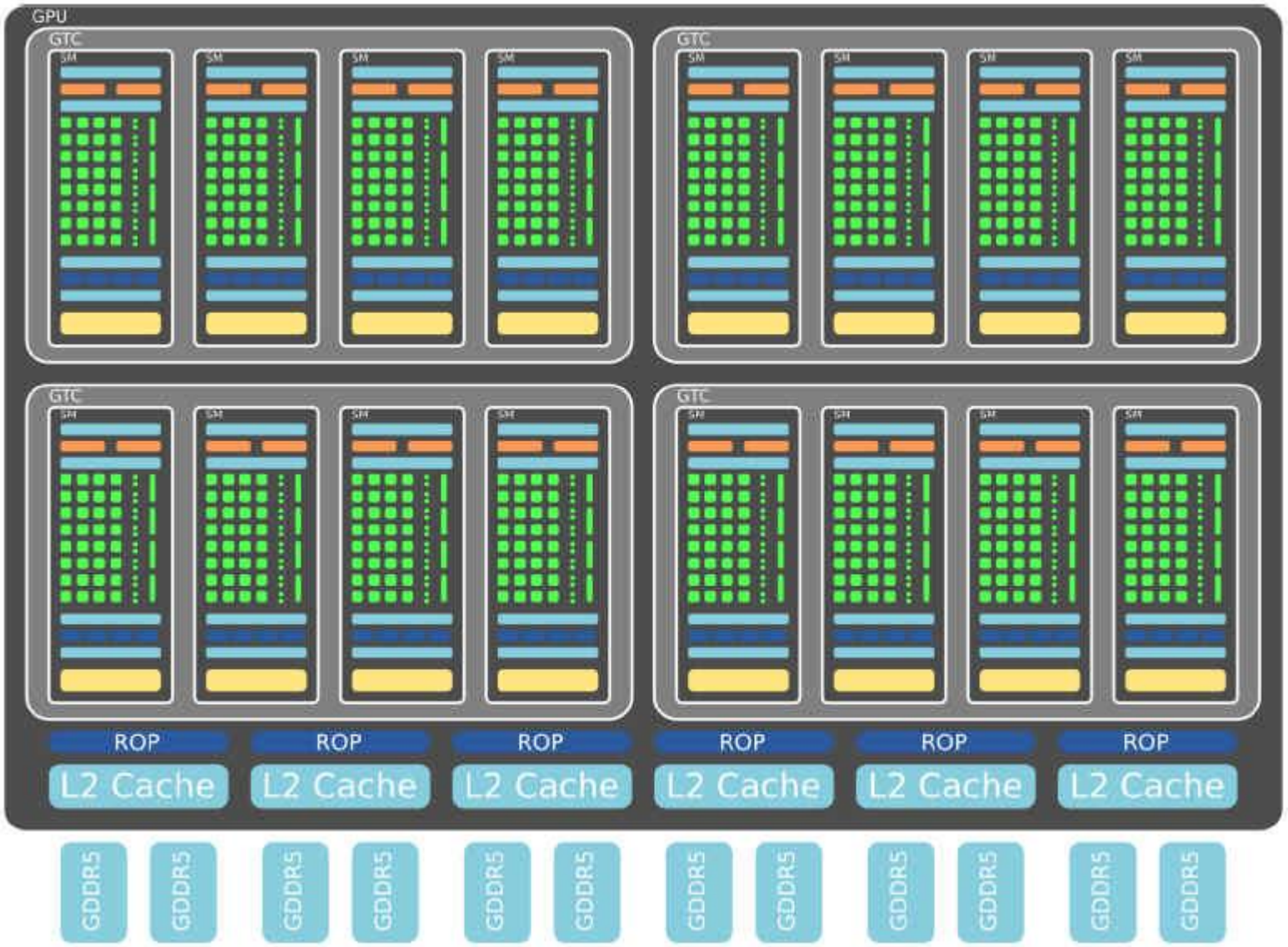
- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files

- 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

Example

- Multiply two vectors of length 8192
  - —Code that works over all elements is the grid
  - —Thread blocks break this down into manageable sizes
    - 512 threads per block
  - —SIMD instruction executes 32 elements at a time
  - —Thus grid size = 16 blocks
  - —Block is analogous to a strip-mined vector loop with vector length of 32
  - —Block is assigned to a *multithreaded SIMD processor* by the *thread block scheduler*
  - —Current-generation GPUs (Fermi) have 7-15 multithreaded SIMD processors

Fermi, 512 Processing Elements (PEs)



CUDA Definition

Type	More descriptive name	Official CUDA/ NVIDIA term	Book definition and AMD and OpenCL terms	Official CUDA/NVIDIA definition
Program abstractions	Vectorizable loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more "Thread Blocks" (or bodies of vectorized loop) that can execute in parallel.	A grid is an array of thread blocks that can execute concurrently, sequentially, or a mixture.
	Body of Vectorized loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. These SIMD Threads can communicate via Local Memory.	A thread block is an array of CUDA Threads that execute concurrently together and can cooperate and communicate via Shared Memory and barrier synchronization. A Thread Block has a Thread Block ID within its Grid.
	Sequence of SIMD Lane operations	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask.	A CUDA Thread is a lightweight thread that executes a sequential program and can cooperate with other CUDA Threads executing in the same Thread Block. A CUDA Thread has a thread ID within its Thread Block.

CUDA Definition (Cont'd)

Type	More descriptive name	Official CUDA/ NVIDIA term	Book definition	Official CUDA/NVIDIA definition
Machine object	A Thread of SIMD instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results are stored depending on a per-element mask.	A warp is a set of parallel CUDA Threads (e.g., 32) that execute the same instruction together in a multithreaded SIMT/SIMD Processor.
	SIMD instruction	PTX instruction	A single SIMD instruction executed across the SIMD Lanes.	A PTX instruction specifies an instruction executed by a CUDA Thread.

CUDA Definition (Cont'd)

Type	More descriptive name	Official CUDA/ NVIDIA term	Book definition	Official CUDA/NVIDIA definition
Processing hardware	Multithreaded SIMD processor	Streaming multi-processor	Multithreaded SIMD Processor that executes thread of SIMD instructions, independent of other SIMD Processors.	A streaming multiprocessor (SM) is a multithreaded SIMT/ SIMD Processor that executes warps of CUDA Threads. A SIMT program specifies the execution of one CUDA Thread, rather than a vector of multiple SIMD Lanes.
	Thread block scheduler	Giga thread engine	Assigns multiple bodies of vectorized loop to multithreaded SIMD Processors.	Distributes and schedules thread blocks of a grid to streaming multiprocessors as resources become available.
	SIMD Thread scheduler	Warp scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.	A warp scheduler in a streaming multiprocessor schedules warps for execution when their next instruction is ready to execute.
	SIMD Lane	Thread processor	Hardware SIMD Lane that executes the operations in a thread of SIMD instructions on a single element. Results are stored depending on mask.	A thread processor is a datapath and register file portion of a streaming multiprocessor that executes operations for one or more lanes of a warp.

CUDA Definition (Cont'd)

Type	More descriptive name	Official CUDA/ NVIDIA term	Book definition	Official CUDA/NVIDIA definition
Memory hardware	GPU Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.	Global memory is accessible by all CUDA Threads in any thread block in any grid; implemented as a region of DRAM, and may be cached.
	Private Memory	Local Memory	Portion of DRAM memory private to each SIMD Lane.	Private "thread-local" memory for a CUDA Thread; implemented as a cached region of DRAM.
	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.	Fast SRAM memory shared by the CUDA Threads composing a thread block, and private to that thread block. Used for communication among CUDA Threads in a thread block at barrier synchronization points.
	SIMD Lane registers	Registers	Registers in a single SIMD Lane allocated across body of vectorized loop.	Private registers for a CUDA Thread; implemented as multithreaded register file for certain lanes of several warps for each thread processor.

Terminology

- *Threads of SIMD instructions*
  - —Each has its own PC
  - —Thread scheduler uses scoreboard to dispatch
  - —No data dependencies between threads!
  - —Keeps track of up to 48 threads of SIMD instructions
    - Hides memory latency
- Thread block scheduler schedules blocks to SIMD processors
- Within each SIMD processor:
  - —32 SIMD lanes
  - —Wide and shallow compared to vector processors

Example

- NVIDIA GPU has 32,768 registers
  - —Divided into lanes
  - —Each SIMD thread is limited to 64 registers
  - —SIMD thread has up to:
    - 64 vector registers of 32 32-bit elements
    - 32 vector registers of 32 64-bit elements
  - —Fermi has 16 physical SIMD lanes, each containing 2048 registers

NVIDIA Instruction Set Arch.

- ISA is an abstraction of the hardware instruction set
  - —"Parallel Thread Execution (PTX)"
  - —Uses virtual registers
  - —Translation to machine code is performed in software
  - —Example:

```
shl.s32         R8, blockIdx, 9    ; Thread Block ID * Block size (512 or 29)
add.s32         R8, R8, threadIdx  ; R8 = i = my CUDA thread ID
ld.global.f64   RD0, [X+R8]        ; RD0 = X[i]
ld.global.f64   RD2, [Y+R8]        ; RD2 = Y[i]
mul.f64 R0D, RD0, RD4              ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, RD0, RD2              ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0          ; Y[i] = sum (X[i]*a + Y[i])
```

Conditional Branching

- Like vector architectures, GPU branch hardware uses internal masks
- Also uses
  - Branch synchronization stack
    - Entries consist of masks for each SIMD lane
    - I.e. which threads commit their results (all threads execute)
  - Instruction markers to manage when a branch diverges into multiple execution paths
    - Push on divergent branch
  - …and when paths converge
    - Act as barriers
    - Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer

Example

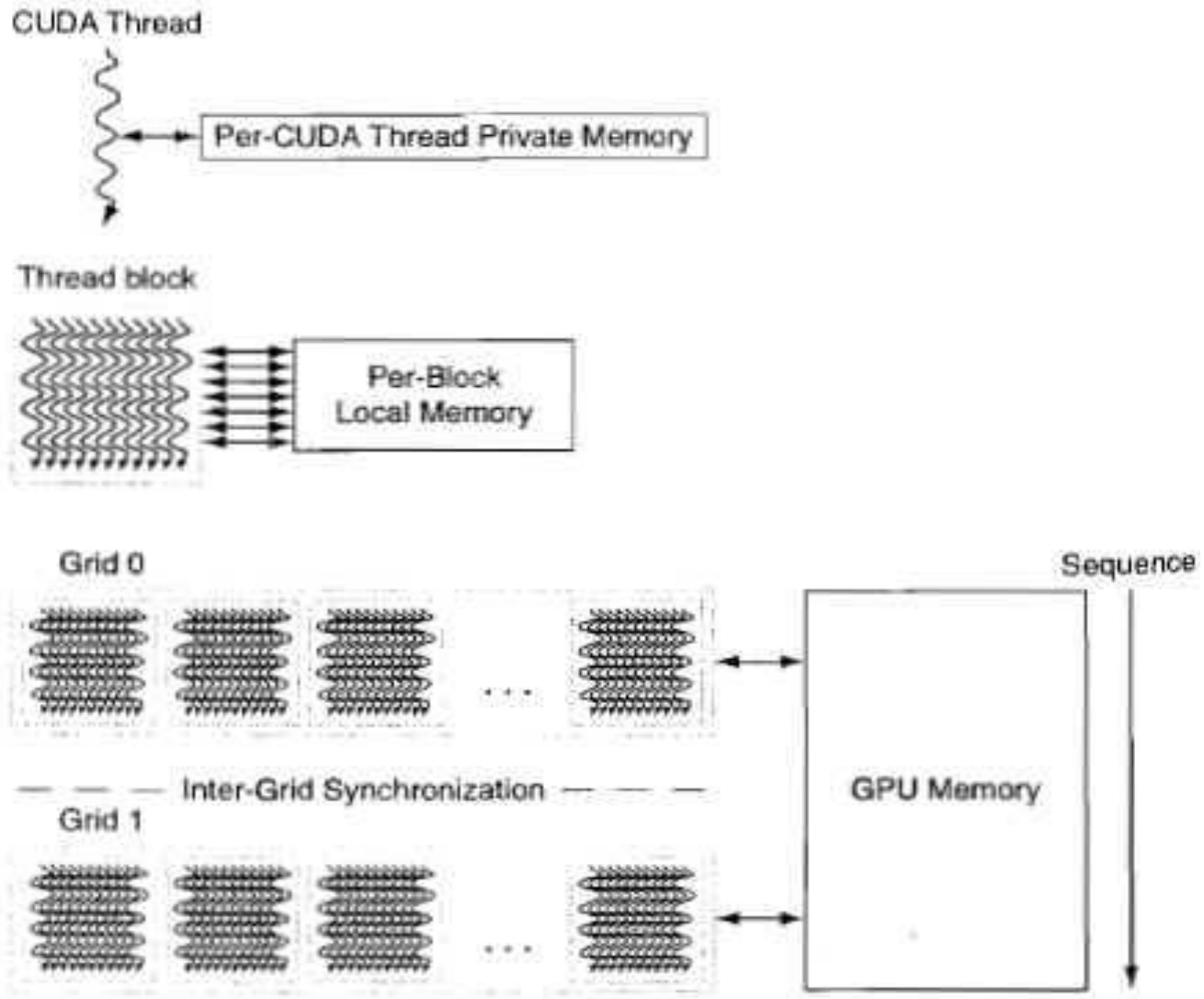
```
if (X[i] != 0)
        X[i] = X[i] – Y[i];
else X[i] = Z[i];

    ld.global.f64     RD0, [X+R8]              ; RD0 = X[i]
    setp.neq.s32      P1, RD0, #0              ; P1 is predicate register 1
    @!P1, bra         ELSE1, *Push             ; Push old mask, set new mask bits
                                               ; if P1 false, go to ELSE1
    ld.global.f64     RD2, [Y+R8]              ; RD2 = Y[i]
    sub.f64           RD0, RD0, RD2            ; Difference in RD0
    st.global.f64     [X+R8], RD0              ; X[i] = RD0
    @P1, bra          ENDIF1, *Comp            ; complement mask bits
                                               ; if P1 true, go to ENDIF1
ELSE1:              ld.global.f64 RD0, [Z+R8]  ; RD0 = Z[i]
                    st.global.f64 [X+R8], RD0  ; X[i] = RD0
ENDIF1:  <next instruction>, *Pop       ; pop to restore old mask
```

NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of off-chip DRAM
  - "Private memory"
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory
  - Host can read and write GPU memory

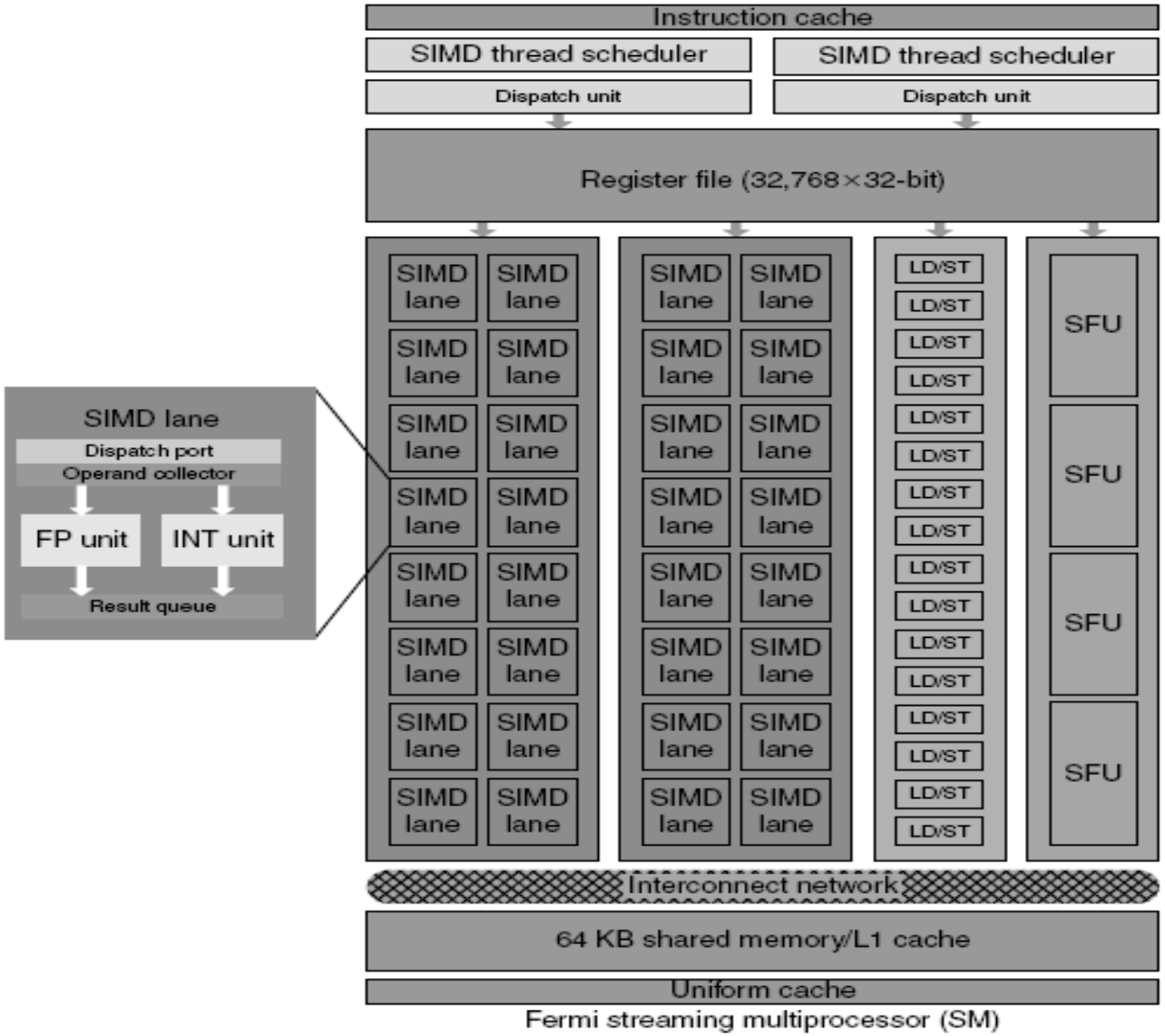
NVIDIA GPU Memory Structures



Fermi Architecture Innovations

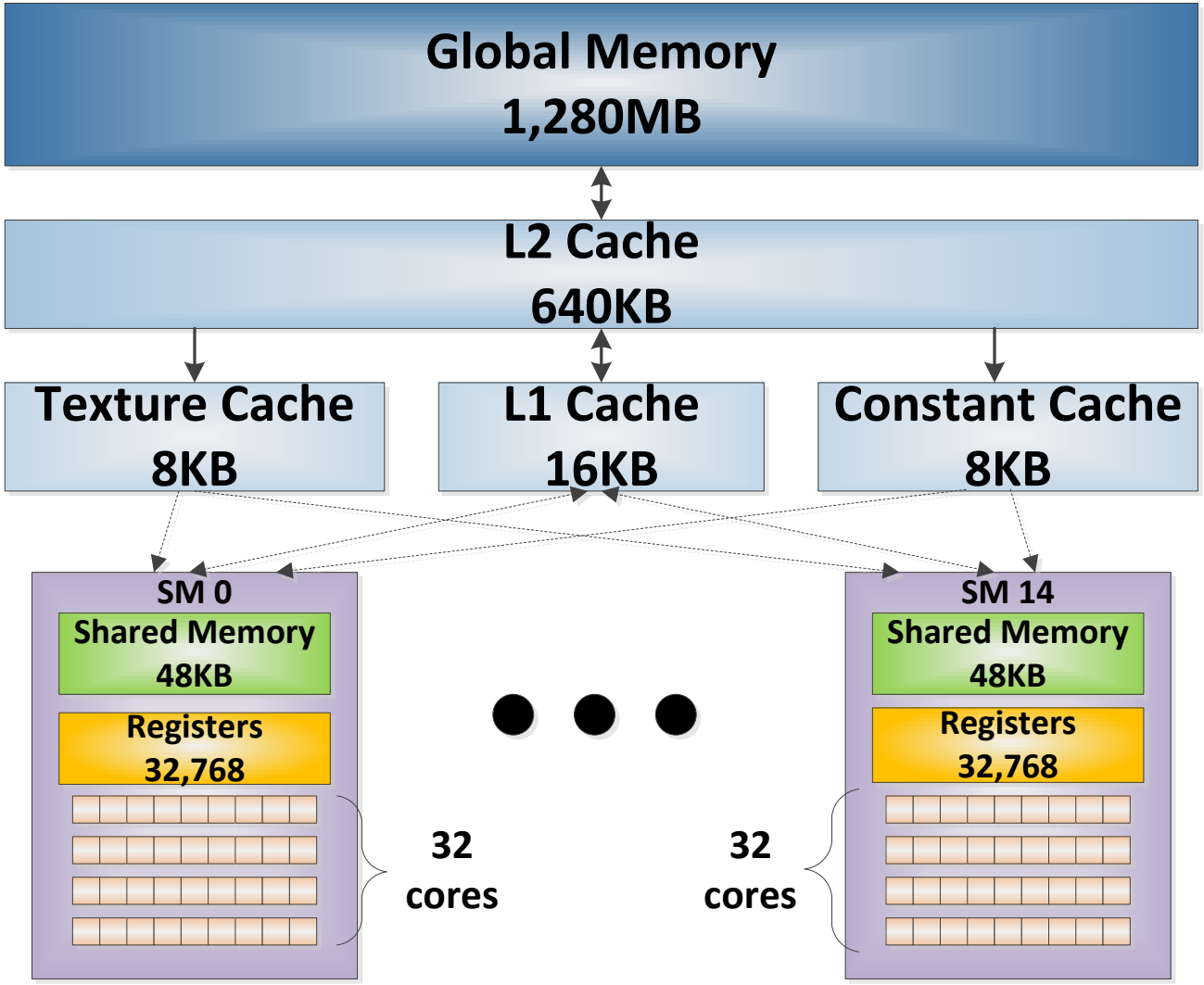
- Each SIMD processor has
  - Two SIMD thread schedulers, two instruction dispatch units
  - 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
  - Thus, two threads of SIMD instructions are scheduled every two clock cycles
- Fast double precision
- Caches for GPU memory
- 64-bit addressing and unified address space
- Error correcting codes
- Faster context switching
- Faster atomic instructions

Fermi Multithreaded SIMD Proc.



Fermi streaming multiprocessor (SM)

GTX570 GPU



Up to 1536
Threads/SM

Vector Processors vs. GPU

- Multiple functional units as opposed to deeply pipelined fewer functional units of Vector processor!
- Two level scheduling:
  - thread block scheduler and thread scheduler
- GPU (32-wide thread of SIMD instructions, 16 lanes) = Vector (16 lanes with vector length of 32) = 2 chimes

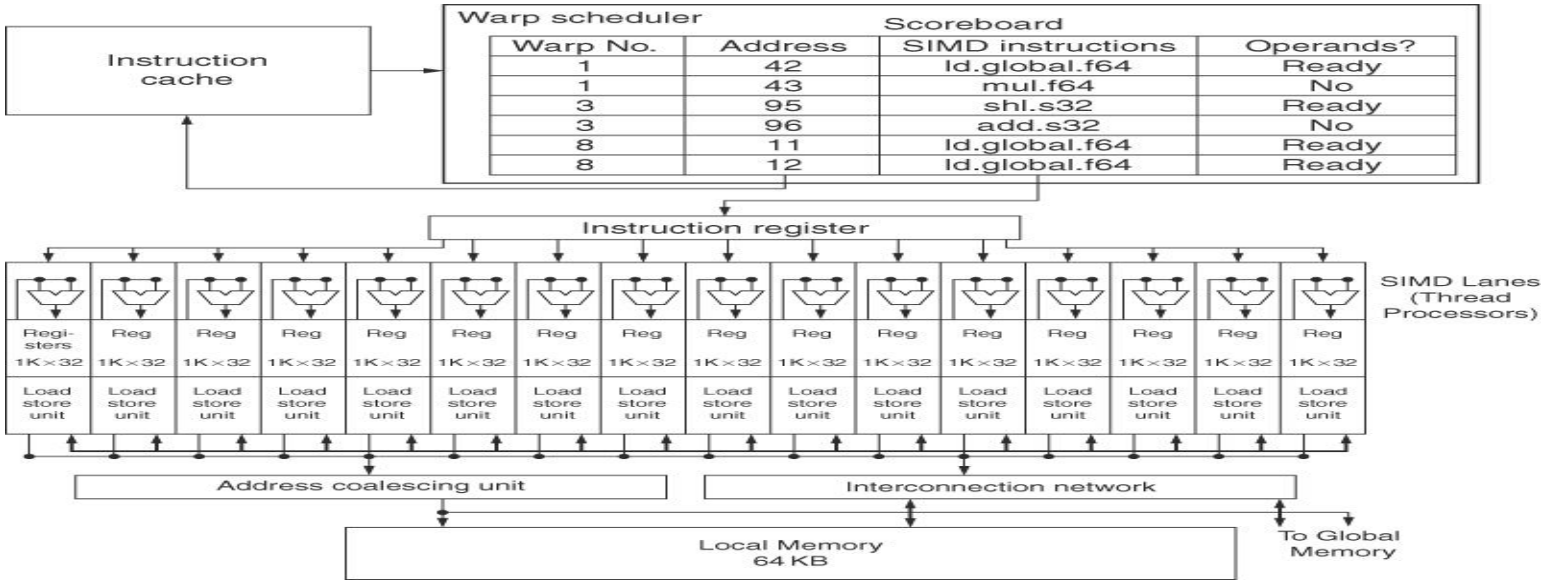
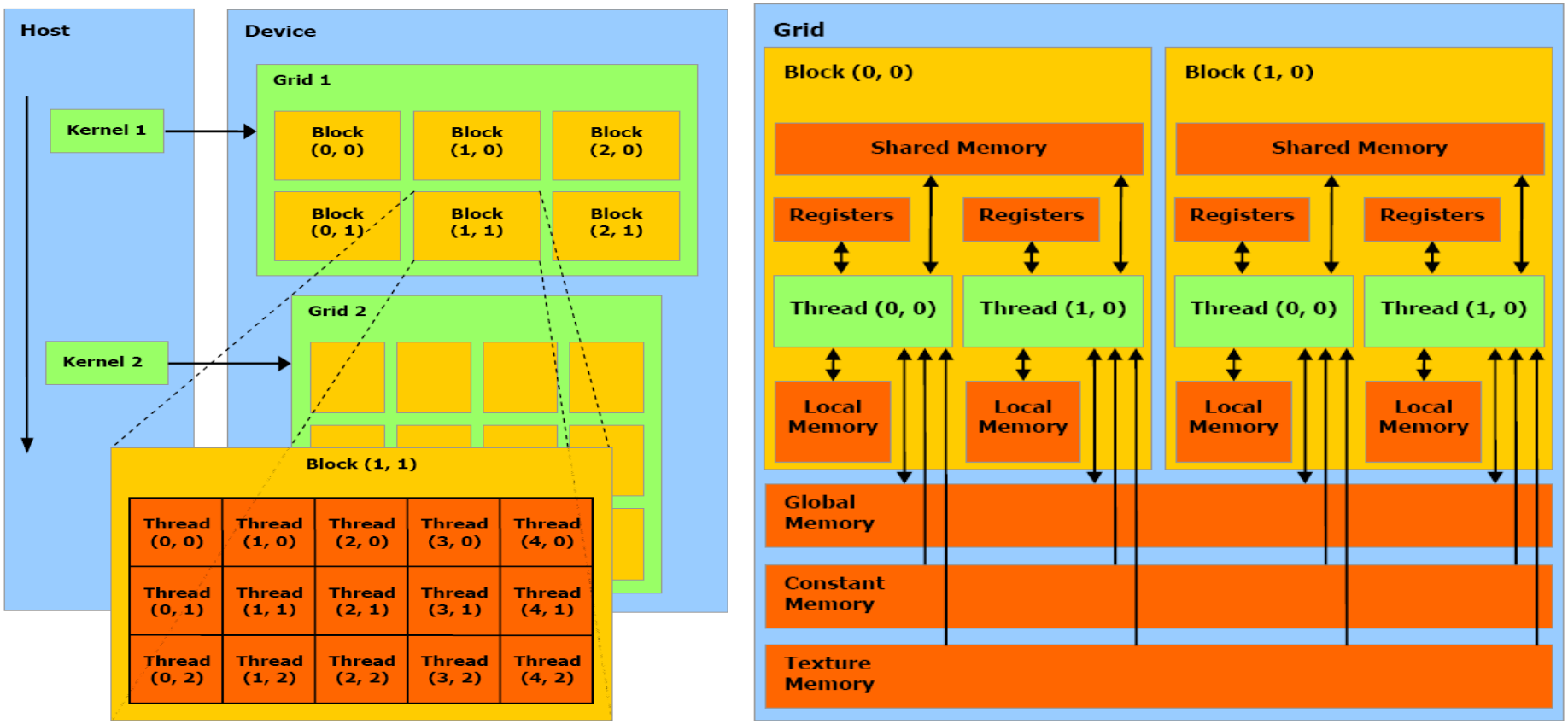


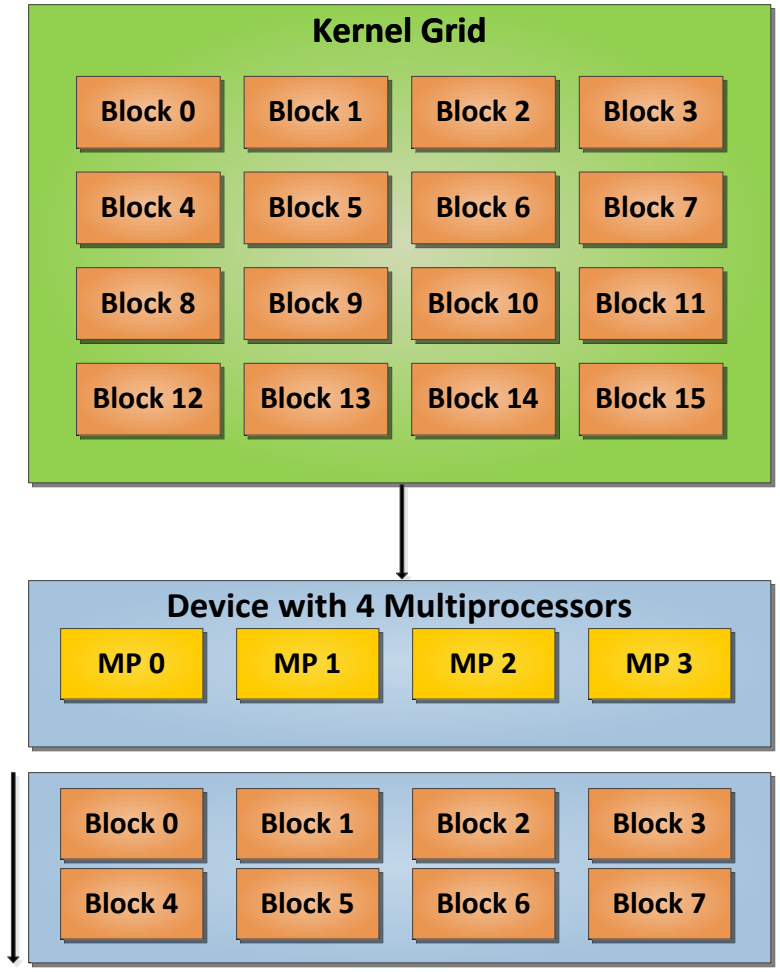
Figure 4.14 Simplified block diagram of a Multithreaded SIMD Processor. It has 16 SIMD lanes. The SIMD Thread Scheduler has, say, 48 independentthreads of SIMD instructions that it schedules with a table of 48 PCs.

GTX570 GPU



- 32 threads within a block work collectively
  - ✓ Memory access optimization, latency hiding

GTX570 GPU



- Up to 1024 Threads/Block and 8 Active Blocks per SM

Vector Processors vs. GPU

- Grid and Thread Block are abstractions for programmer
- SIMD Instruction on GPU = Vector instruction on Vector
- SIMD instructions of each thread is 32 element wide
  - thread block with 32 threads =
  - Strip-minded vector loop with a vector length of 32
- Each SIMD-thread is limited to no more than 64 registers
  - 64 vector registers , each with 32-bit 32 elements
  - or 32 vector registers, each with 64-bit 32 elements
  - 32,768 threads for 16 SIMD-Lanes (2048/lane)

Vector Processors vs. GPU (Cont'd, Loop)

- Both rely on independent loop iterations
- GPU:
  - Each iteration becomes a thread on the GPU
  - Programmer specifies parallelism
    - grid dimensions and threads/block
  - Hardware handles parallel execution and thread management
  - Trick: have 32 threads/block, create many more threads per SIMD multi-processor to hide memory latency

Vector Processors vs. GPU (Cont'd)

- Conditional Statements
  - Vector:
    - mask register part of the architecture
    - Rely on compiler to manipulate mask register
  - GPU:
    - Use hardware to manipulate internal mask registers
    - Mask register not visible to software
  - Both spend time to execute masking
- Gather-Scatter
  - GPU:
    - all loads are gathers and stores are scatters
    - Programmer should make sure that all addresses in a gather or scatter are adjacent locations

Vector Processors vs. GPU (Cont'd)

- Multithreading
  - GPU: yes
  - Vector: no
- Lanes
  - GPU: 16-32
  - A SIMD thread of 32 element wide: 1-2 chime
  - Vector 2-8
  - Vector length of 32: chime to 4 - 16
- Registers
  - GPU (Each SIMD thread): 64 registers with 32 elements
  - Vector: 8 vector registers with 64 elements
- Latency
  - Vector: deeply pipelined, once per vector load/store
  - GPU: hides latency with multithreading

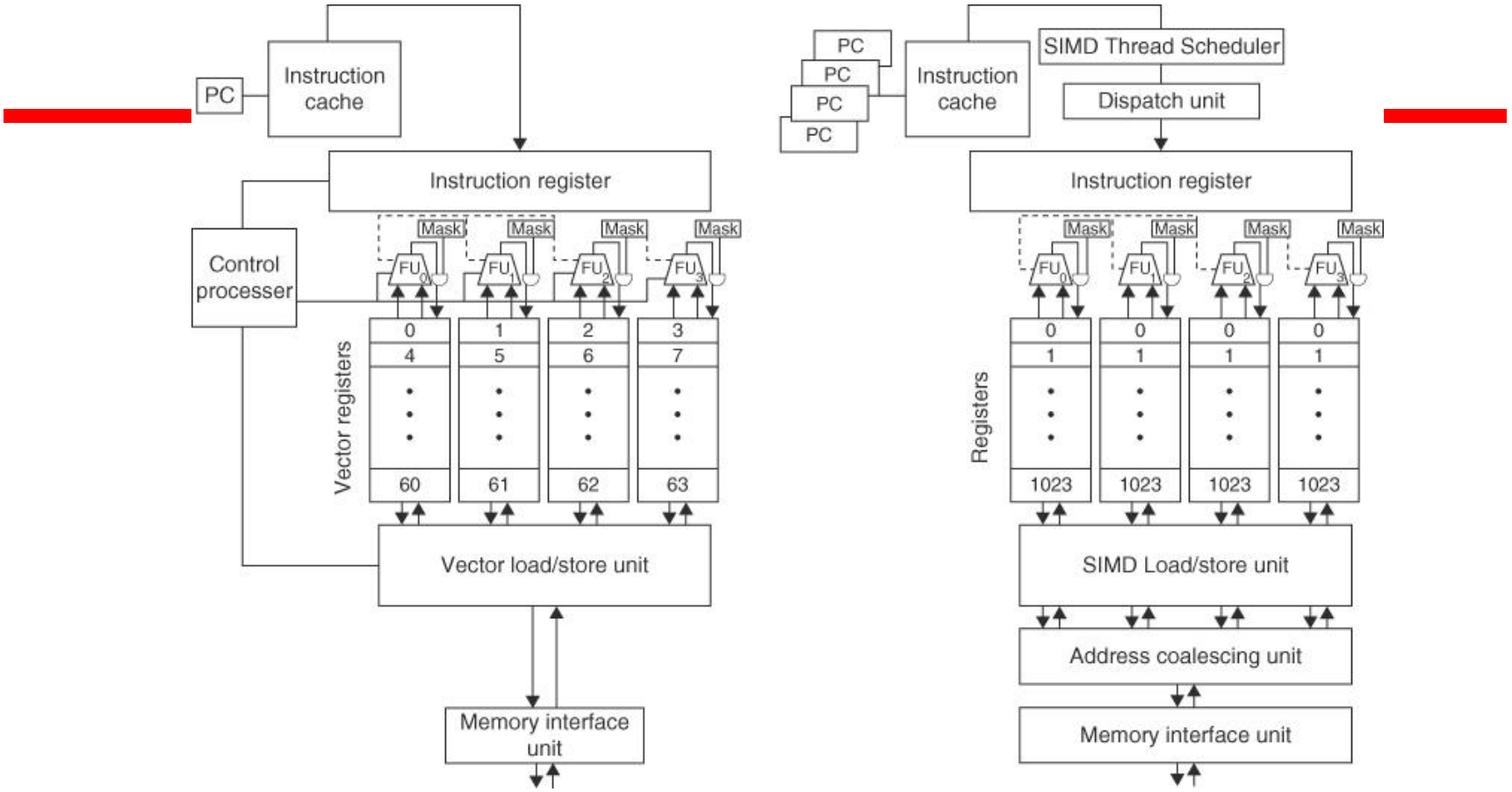


Figure 4.22 A vector processor with four lanes on the left and a multithreaded SIMD Processor of a GPU with four SIMD Lanes on the right. (GPUs typically have 8 to 16 SIMD Lanes.) The control processor supplies scalar operands for scalar-vector operations, increments addressing for unit and non-unit stride accesses to memory, and performs other accounting-type operations. Peak memory performance only occurs in a GPU when the Address Coalescing unit can discover localized addressing. Similarly, peak computational performance occurs when all internal mask bits are set identically. Note that the SIMD Processor has one PC per SIMD thread to help with multithreading.

Loop-Level Parallelism

- Focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations
  - Loop-carried dependence

- Example 1:

```
for (i=999; i>=0; i=i-1)
        x[i] = x[i] + s;
```

- No loop-carried dependence

Loop-Level Parallelism

- Example 2:

```
for (i=0; i<100; i=i+1) {
        A[i+1] = A[i] + C[i]; /* S1 */
        B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

- S1 and S2 use values computed by S1 in previous iteration
- S2 uses value computed by S1 in same iteration

Loop-Level Parallelism

- Example 3:

```
for (i=0; i<100; i=i+1) {
        A[i] = A[i] + B[i]; /* S1 */
        B[i+1] = C[i] + D[i]; /* S2 */
}
```

- S1 uses value computed by S2 in previous iteration but dependence is not circular so loop is parallel
- Transform to:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
        B[i+1] = C[i] + D[i];
        A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

Loop-Level Parallelism

- Example 4:

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

- Example 5:

```
for (i=1;i<100;i=i+1)  {
        Y[i] = Y[i-1] + Y[i];
}
```

Finding dependencies

- Assume indices are affine:
  - $a \times i + b$ (i is loop index)

- Assume:
  - Store to $a \times i + b$, then
  - Load from $c \times i + d$
  - i runs from m to n
  - Dependence exists if:
    - Given j, k such that $m \leq j \leq n$, $m \leq k \leq n$
    - Store to $a \times j + b$, load from $a \times k + d$, and $a \times j + b = c \times k + d$

Finding dependencies

- Generally cannot determine at compile time
- Test for absence of a dependence:
  - GCD test:
    - If a dependency exists, GCD(*c*,*a*) must evenly divide (*d*-*b*)

- Example:

```
for (i=0; i<100; i=i+1) {
   X[2*i+3] = X[2*i] * 5.0;
}
```

Finding dependencies

- Example 2:

```
for (i=0; i<100; i=i+1) {
    Y[i] = X[i] / c; /* S1 */
    X[i] = X[i] + c; /* S2 */
    Z[i] = Y[i] + c; /* S3 */
    Y[i] = c - Y[i]; /* S4 */
}
```

- Watch for antidependencies and output dependencies

Finding dependencies

- Example 2:

```
for (i=0; i<100; i=i+1) {
    Y[i] = X[i] / c; /* S1 */
    X[i] = X[i] + c; /* S2 */
    Z[i] = Y[i] + c; /* S3 */
    Y[i] = c - Y[i]; /* S4 */
}
```

- Watch for antidependencies and output dependencies

Reductions

- Reduction Operation:

```
for (i=9999; i>=0; i=i-1)
        sum = sum + x[i] * y[i];
```

- Transform to…

```
for (i=9999; i>=0; i=i-1)
        sum [i] = x[i] * y[i];
for (i=9999; i>=0; i=i-1)
        finalsum = finalsum + sum[i];
```

- Do on p processors:

```
for (i=999; i>=0; i=i-1)
        finalsum[p] = finalsum[p] + sum[i+1000*p];
```

- Note: assumes associativity!

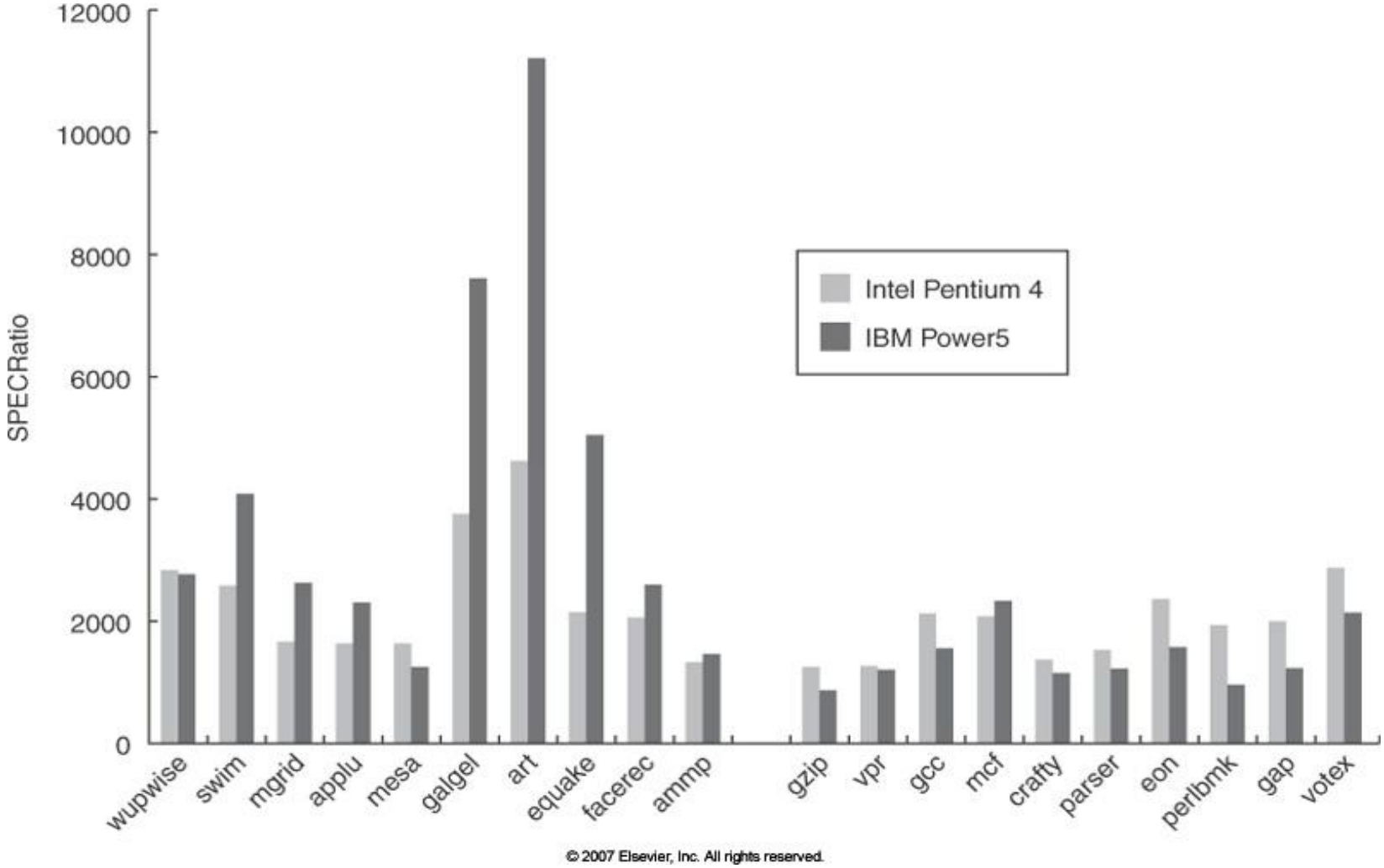
Fallacy 1

- It is easy to predict the performance and energy efficiency of two different versions of the same instruction set architecture, if we hold the technology constant.

Fallacy 2

- Processors with lower CPIs will always be faster.
- Processors with faster clock rates will always be faster
- Although a lower CPI is certainly better, sophisticated pipelines typically have slower clock rates than processors with simple pipelines
- In APs with limited ILP or where the parallelism cannot be exploited by the hardware resources, the faster clock rate often wins
- But, when significant ILP exists, a processor that exploits lots of ILP may be better

Fallacy (Cont'd)



Pitfall

- Sometimes bigger and dumber is better
- Advanced pipelines have focused on novel and increasingly sophisticated schemes for improving CPI
  - The 21264 uses a 29 Kbits sophisticated tournament predictor
  - the earlier 21164 uses a simple 2-bit predictor with 2K entries (or a total of 4 Kbits)
- For the SPEC95 benchmarks, the more sophisticated branch predictor of the 21264 outperforms the simpler 2-bit scheme on all but one benchmark

Pitfall (Cont'd)

- On average, for SPECInt95, the 21264 has 11.5 mispredictions per 1000 instructions committed while the 21164 has about 16.5 mispredictions
- Somewhat surprisingly, the simpler 2-bit scheme works better for the transaction processing workload than the sophisticated 21264 scheme (17 mispredictions vs. 19 per 1000 completed instructions)!
- How can a predictor with less than 1/7 the number of bits and a much simpler scheme actually work better?
  - —The answer lies in the structure of the workload

Pitfall (Cont'd)

- The transaction processing workload has a very large code size with a large branch frequency
  - —The ability of the 21164 predictor to hold twice as many branch predictions based on purely local behavior (2K vs. the 1K local predictor in the 21264) seems to provide a slight advantage
- Different applications can produce different behaviors