
Parallel Computing Platforms: Control Structures and Memory Hierarchy

John Mellor-Crummey

**Department of Computer Science
Rice University**

johnmc@rice.edu

Topics for Today

- **SIMD and MIMD control structure**
- **Memory hierarchy and performance**

Parallel Computing Platforms

A parallel computing platform must specify

- concurrency = control structure**
- interaction between concurrent tasks = communication model**

Control Structure of Parallel Platforms

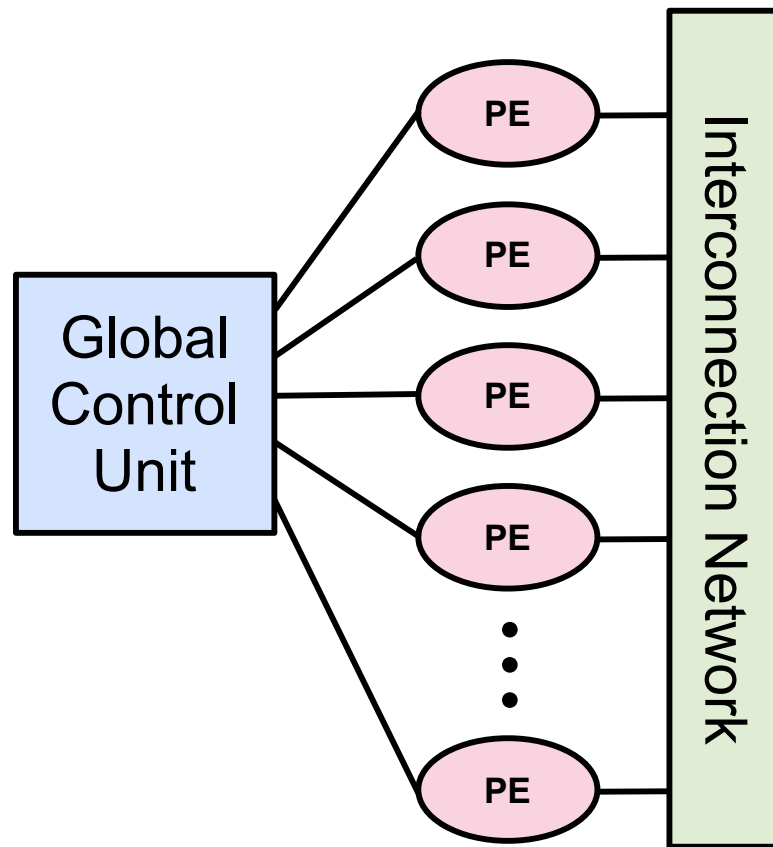
Parallelism ranges from instructions to processes

- **Processor control structure alternatives**
 - work independently
 - operate under the centralized control of a single control unit
- **MIMD**
 - **Multiple Instruction streams**
 - each hardware thread has its own control unit
 - each hardware thread can execute different instructions
 - **Multiple Data streams**
 - each thread can work on its own data
- **SIMD**
 - **Single Instruction stream**
 - single control unit dispatches the same instruction to processing elements
 - **Multiple Data streams**
 - processing elements work on their own data

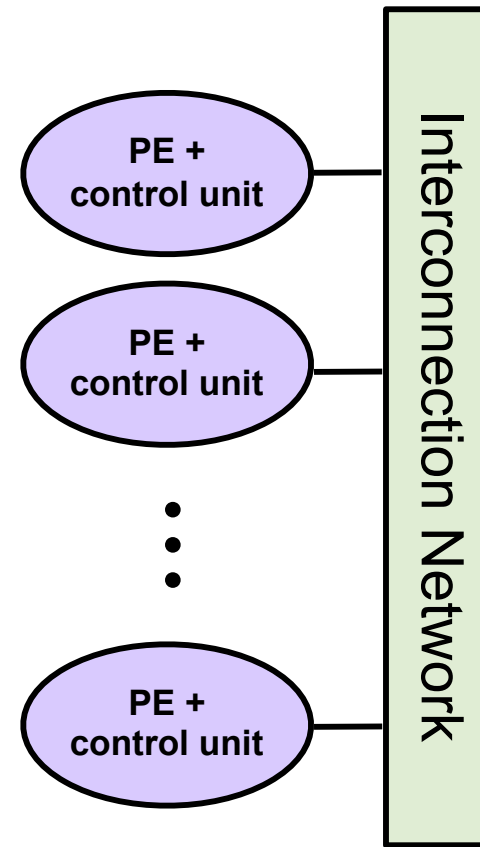
Control Structure of Parallel Platforms - II

- **SIMT**
 - **Single Instruction stream**
 - single control unit dispatches the same instruction to processing element
 - **Multiple Threads**
- **SIMT features that SIMD lacks**
 - single instruction, multiple register sets
 - SIMT processing elements have a separate register set per thread
 - single instruction, multiple flow paths
 - one can write if statement blocks that contain more than a single operation. some processors will execute the code, others will no-op.

SIMD and MIMD Processors



SIMD architecture



MIMD architecture

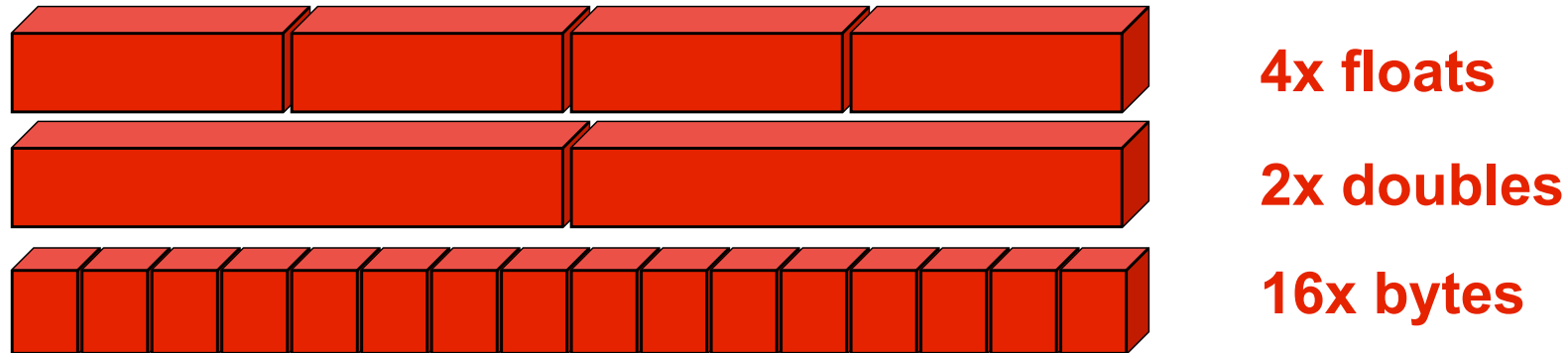
PE = Processing Element

SIMD Control

- **SIMD excels for computations with regular structure**
 - media processing, scientific kernels (e.g., linear algebra, FFT)
- **Activity mask**
 - per PE predicated execution: turn off operations on certain PEs
 - each PE tests own conditional and sets own activity mask
 - PE can conditionally perform operation predicated on mask value

Example: 128-bit SIMD Vectors

- Data types: anything that fits into 16 bytes, e.g.,



- Instructions operate in parallel on data in this 16 byte register
 - add, multiply etc.
- Data bytes must be **contiguous in memory** and **aligned**
- Additional instructions needed for
 - masking data
 - moving data from one part of a register to another

Computing with SIMD Vector Units

- **Scalar processing**
—one operation produces one result

X

+

Y

X + Y

- **SIMD vector units**
—one operation produces multiple results

X

x3 x2 x1 x0

+

Y

y3 y2 y1 y0

X + Y

x3+y3 x2+y2 x1+y1 x0+y0

Executing a Conditional on a SIMD Processor

conditional statement

```
if (A == 0)
  then C = B
  else C = B/A
```

initial values

A	0
B	5
C	0

Processor 0

A	4
B	8
C	0

Processor 1

A	2
B	2
C	0

Processor 2

A	0
B	7
C	0

Processor 3

execute
"then" branch

A	0
B	5
C	5

Processor 0

A	4
B	8
C	0

Processor 1

A	1
B	2
C	0

Processor 2

A	0
B	7
C	7

Processor 3

execute
"else" branch

A	0
B	5
C	5

Processor 0

A	4
B	8
C	2

Processor 1

A	2
B	2
C	1

Processor 2

A	0
B	7
C	7

Processor 3

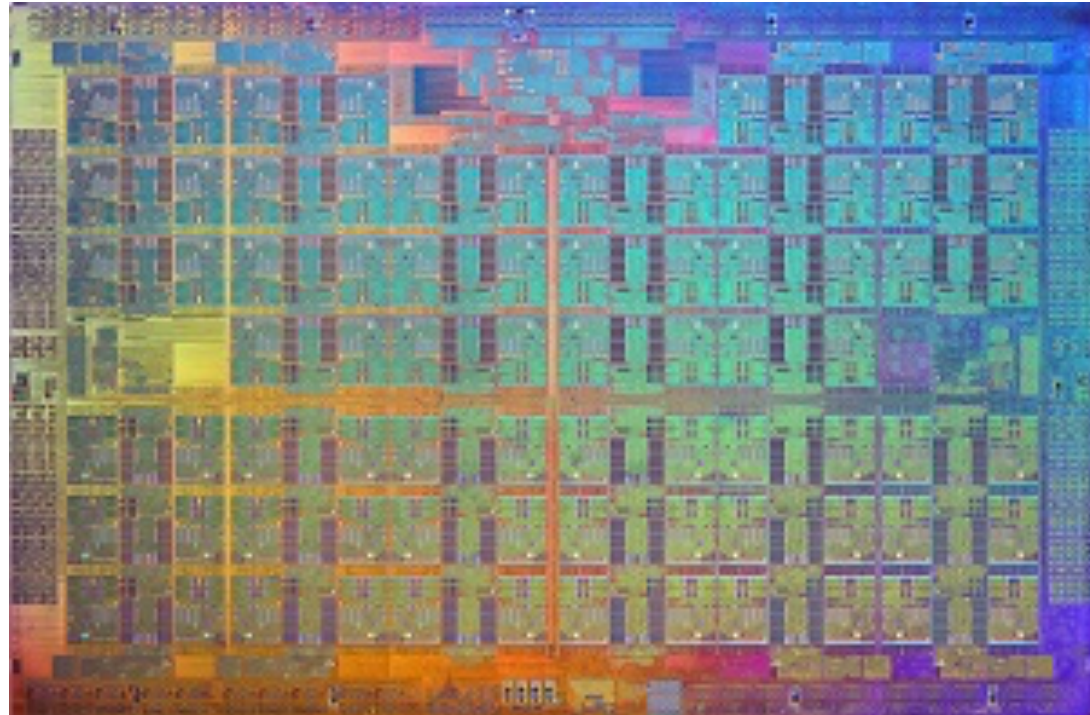
SIMD Examples

- Previously: SIMD computers
 - e.g., Connection Machine CM-1/2, and MasPar MP-1/2
 - CM-1 (1980s): 65,536 1-bit processors
- Today: SIMD units or co-processors (AKA accelerators)
 - vector units
 - SSE/2/3/4 - Streaming SIMD Extensions
 - 8 128-bit vector registers
 - 128 bits as 8-bit chars, 16-bit words, 32/64-bit int and float
 - AVX - Advanced Vector Extensions
 - 16 256-bit vector registers in Intel and AMD processors since 2011
 - 256 bits as 8-bit chars, 16-bit words, 32/64-bit int and float
 - 32 512-bit vector registers in Intel Xeon Phi
 - 512 bits as 8-bit chars, 16-bit words, 32/64-bit int and float
 - co-processors
 - ClearSpeed CSX700 array processor (control PE + array of 96 PEs)
 - nVIDIA PASCAL P100 GPGPU

Intel Knight's Landing (2016)

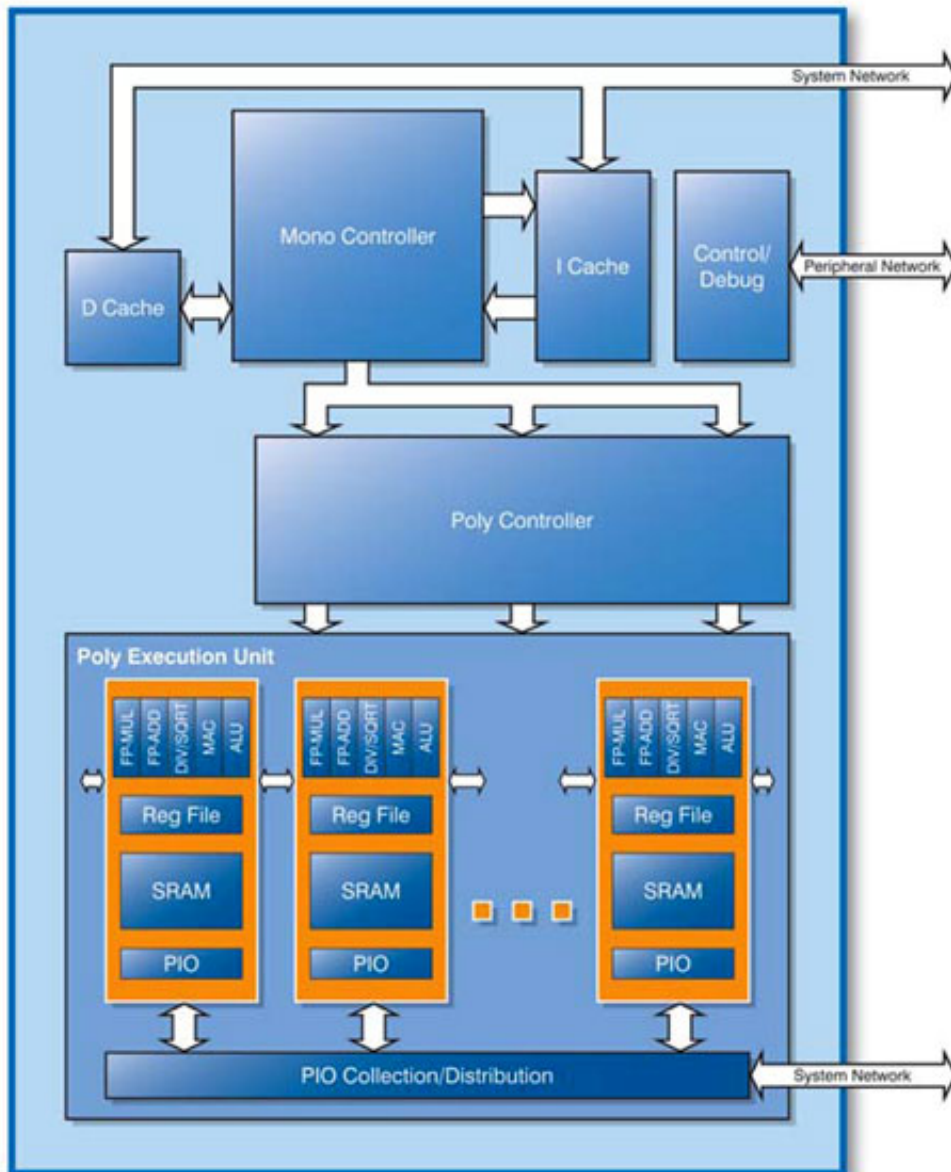
- > 8 billion transistors
- Self-hosted manycore processor
- Up to 72-cores
 - 4 SMT threads per core
 - 32 512-bit vector registers
- Up to 384GB of DDR4-2400 main memory
 - 115GB/s max mem BW
- Up to 16GB of MCDRAM on-package (3D stacked)
 - 400GB/s max mem BW
- 3.46TF double precision

2nd Generation Xeon Phi



SIMD: ClearSpeed MTAP Co-processor

MTAP processor



- **Features**
 - hardware multi-threading
 - asynchronous, overlapped I/O
 - extensible instruction set
- **SIMD core**
 - poly controller
 - poly execution unit
 - array of 192 PEs
 - 64- and 32-bit floating point
 - 250 MHz (key to low power)
 - 96 GFLOP, <15 Watts

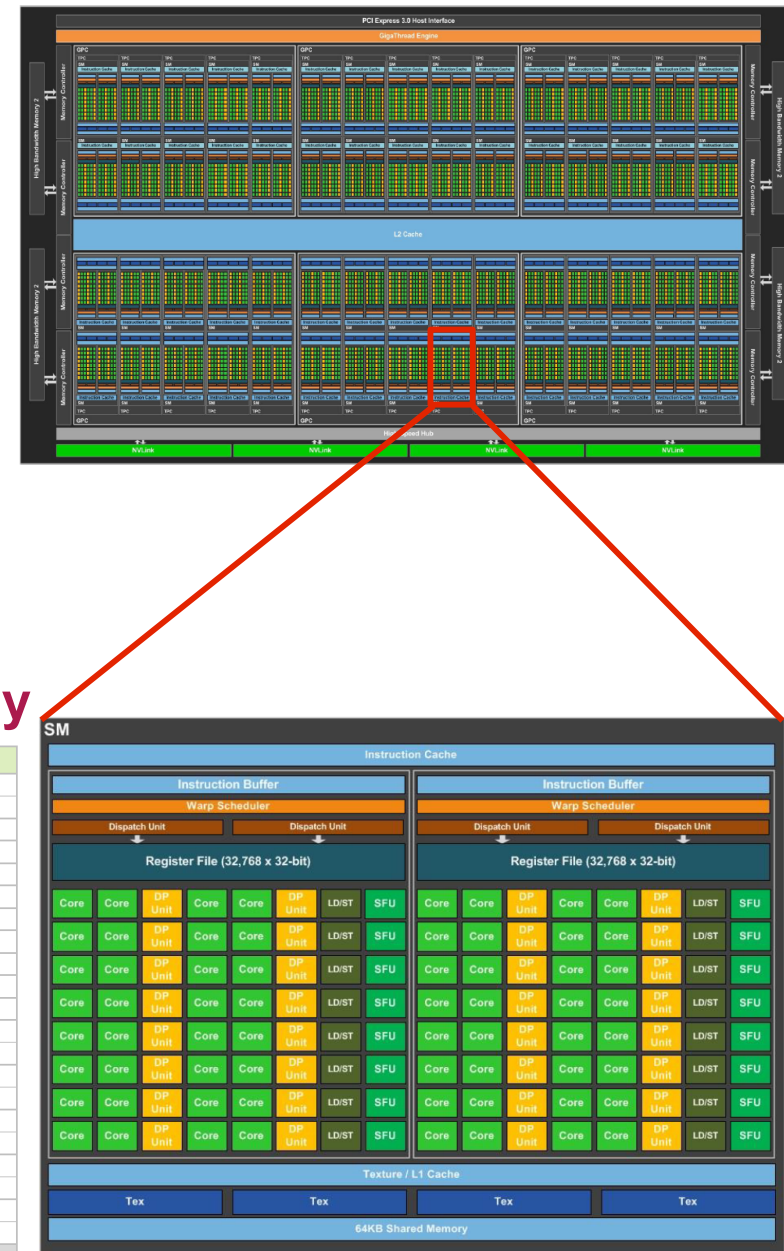
**(CSX700 released June 2008
company delisted in 2009)**

NVIDIA PASCAL P100 (2016)

- 56 Streaming Multiprocessors (SMs)
- Each SM
 - 64 CUDA cores (single precision)
 - fully pipelined FP and INT units
 - IEEE 754-2008; fused multiply add
 - two warp schedulers
 - 32-thread groups (warp)
 - 2 warps issue and execute concurrently
 - 2 inst/warp/cycle
 - 32 DP FP units
 - 16 LS units
 - 16 SFU
- 5.3 TFLOPs DP
300W

Tesla Products	Tesla K40	Tesla M40	Tesla P100
GPU	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)
SMs	15	24	56
TPCs	15	24	28
FP32 CUDA Cores / SM	192	128	64
FP32 CUDA Cores / GPU	2880	3072	3584
FP64 CUDA Cores / SM	64	4	32
FP64 CUDA Cores / GPU	960	96	1792
Base Clock	745 MHz	948 MHz	1328 MHz
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz
Peak FP32 GFLOPs ¹	5040	6840	10600
Peak FP64 GFLOPs ¹	1680	210	5300
Texture Units	240	192	224
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB
TDP	235 Watts	250 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion
GPU Die Size	551 mm ²	601 mm ²	610 mm ²
Manufacturing Process	28-nm	28-nm	16-nm FinFET

¹ The GFLOPs in this chart are based on GPU Boost Clocks.



NVIDIA PASCAL P100 (2016)

- 56 Streaming Multiprocessors (SMs)

- Each SM contains

- 64 CUDA Cores

- full precision FP32

- IEEE 754 double precision FP64

- two 32-bit integer ALUs

- 32-bit integer ALUs

- 2 Texture Units

- 2 Memory Interfaces

- 32 Double Precision FP64 Cores

- 16 Single Precision FP32 Cores

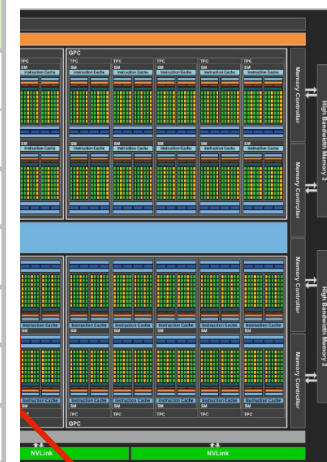
- 16 Single Precision FP32 Cores

- 5.3 TFLOPS (FP32)

300W

Tesla Products	Tesla K40	Tesla M40	Tesla P100
GPU	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)
SMs	15	24	56
TPCs	15	24	28
FP32 CUDA Cores / SM	192	128	64
FP32 CUDA Cores / GPU	2880	3072	3584
FP64 CUDA Cores / SM	64	4	32
FP64 CUDA Cores / GPU	960	96	1792
Base Clock	745 MHz	948 MHz	1328 MHz
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz
Peak FP32 GFLOPs ¹	5040	6840	10600
Peak FP64 GFLOPs ¹	1680	210	5300
Texture Units	240	192	224
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB
TDP	235 Watts	250 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion
GPU Die Size	551 mm ²	601 mm ²	610 mm ²
Manufacturing Process	28-nm	28-nm	16-nm FinFET

¹ The GFLOPS in this chart are based on GPU Boost Clocks.



Short Vectors: The Good and Bad

<pre>for (t = 0; t < T; ++t) { for (i = 0; i < N; ++i) for (j = 1; j < N+1; ++j) S1: C[i][j] = A[i][j] + A[i][j-1]; for (i = 0; i < N; ++i) for (j = 1; j < N+1; ++j) S2: A[i][j] = C[i][j] + C[i][j-1]; }</pre>	<pre>for (t = 0; t < T; ++t) { for (i = 0; i < N; ++i) for (j = 0; j < N; ++j) S3: C[i][j] = A[i][j] + B[i][j]; for (i = 0; i < N; ++i) for (j = 0; j < N; ++j) S4: A[i][j] = B[i][j] + C[i][j]; }</pre>																		
<table><tr><td>Performance:</td><td>AMD Phenom</td><td>1.2 GFlop/s</td></tr><tr><td></td><td>Core2</td><td>3.5 GFlop/s</td></tr><tr><td></td><td>Core i7</td><td>4.1 GFlop/s</td></tr></table>	Performance:	AMD Phenom	1.2 GFlop/s		Core2	3.5 GFlop/s		Core i7	4.1 GFlop/s	<table><tr><td>Performance:</td><td>AMD Phenom</td><td>1.9 GFlop/s</td></tr><tr><td></td><td>Core2</td><td>6.0 GFlop/s</td></tr><tr><td></td><td>Core i7</td><td>6.7 GFlop/s</td></tr></table>	Performance:	AMD Phenom	1.9 GFlop/s		Core2	6.0 GFlop/s		Core i7	6.7 GFlop/s
Performance:	AMD Phenom	1.2 GFlop/s																	
	Core2	3.5 GFlop/s																	
	Core i7	4.1 GFlop/s																	
Performance:	AMD Phenom	1.9 GFlop/s																	
	Core2	6.0 GFlop/s																	
	Core i7	6.7 GFlop/s																	
(a) Stencil code	(b) Non-Stencil code																		

The stencil code (a) has much lower performance than the non-stencil code (b) despite accessing 50% fewer data elements

- Diagram illustrating the state of Vector Registers (VR1 to VR5) and a code snippet:

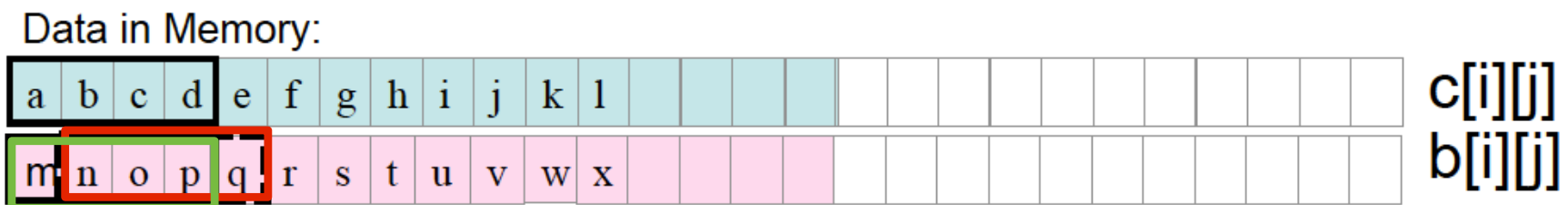
	0	1	2	3
VR1	a	b	c	d
VR2	m	n	o	p
VR3	n	o	p	q
VR4				
VR5				

Code Snippet:

```

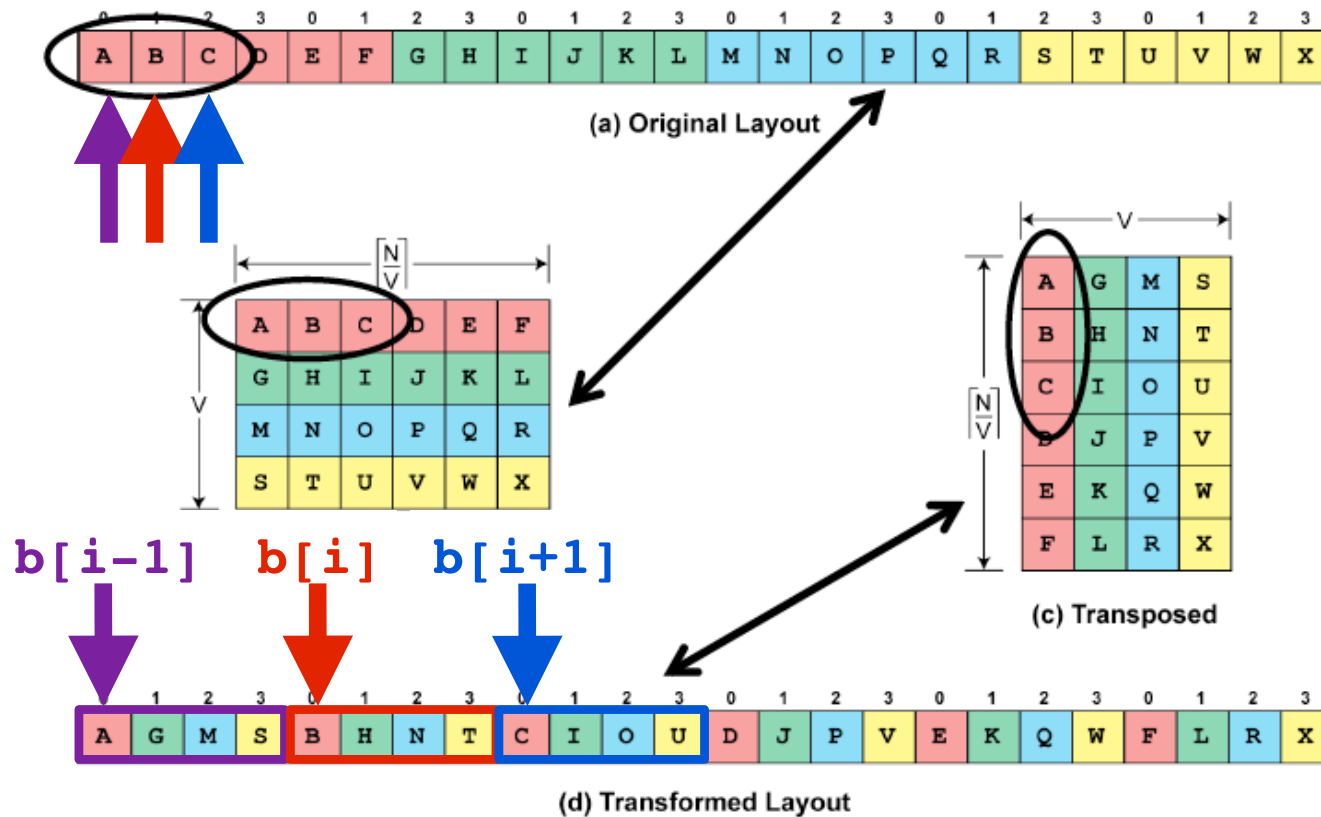
for (i = 0; i < H; ++i)
  for (j = 4; j < W; ++j)
    c[i][j] = b[i][j+1] + b[i][j]
  
```

An arrow points from the code snippet to the red elements (n, o, p, q) in VR3, indicating the data being processed.



- **Stream alignment conflict between $b[i][j+1]$ and $c[i][j]$**

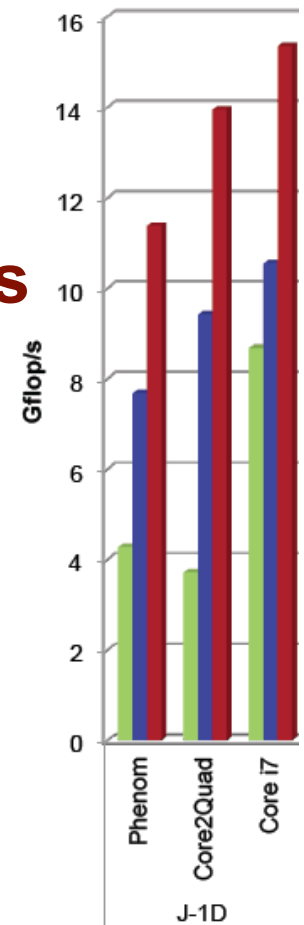
Dimension-lifted Transformation (DLT)



**DLT +
vector
intrinsic**

**DLT +
autovec**

original



- (a) 1D array in memory
- (b) 2D view of same array
- (c) Transposed 2D array brings non-interacting elements into contiguous vectors
- (d) New 1D layout after transformation

Jacobi-1D:
 $a[i] = b[i-1] + b[i] + b[i+1]$

MIMD Processors

Execute different programs on different processors

- **Platforms include current generation systems**
 - **shared memory**
 - multicore laptop
 - workstation with multiple quad core processors
 - Legacy:
 - SGI UV 3000 (up to 256 sockets, each with 8 cores)
 - Cray X1 (up to 8K processors)
 - **distributed memory**
 - clusters (e.g., biou.rice.edu, davinci.rice.edu)
 - Cray XC, IBM Blue Gene
- **SPMD programming paradigm**
 - **Single Program, Multiple Data streams**
 - **same program on different PEs, behavior conditional on thread id**

SIMD vs. MIMD

- **SIMD platforms**

- special purpose: not well-suited for all applications
- custom designed with long design cycles
- less hardware: single control unit
- need less memory: only 1 copy of program
- today
 - SIMD common only for vector units
 - SIMT common for GPUs, e.g., nVIDIA PASCAL

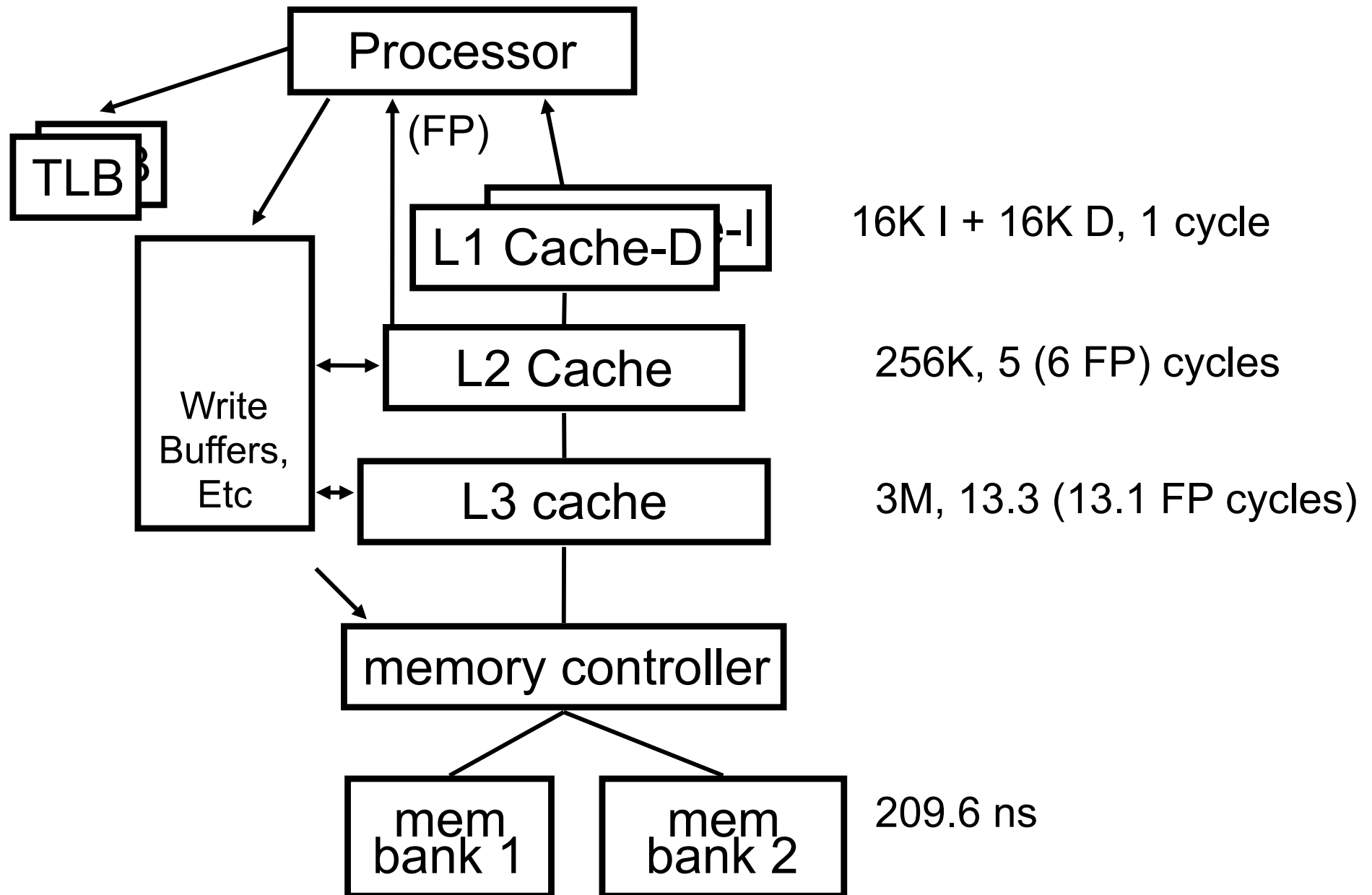
- **MIMD platforms**

- suitable for broad range of applications
- inexpensive: off-the-shelf components + short design cycle
- need more memory: program and OS on each processor

Data Movement and Communication

- **Latency**: How long does a single operation take?
 - measured in microseconds
- **Bandwidth**: What data rate can be sustained?
 - measured in Mbytes or GBytes per second
- These terms can be applied to
 - memory access
 - messaging

A Memory Hierarchy (Itanium 2)



Memory Bandwidth

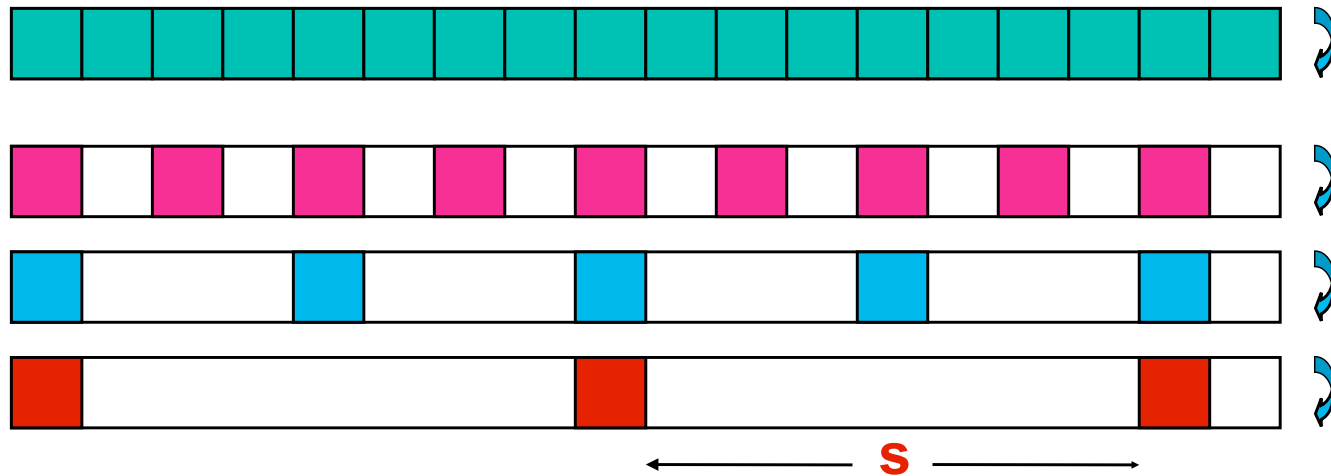
- **Limited by both**
 - the bandwidth of the memory bus
 - the bandwidth of the memory modules
- **Can be improved by increasing the size of memory blocks**
- **Memory system takes L time units to deliver B units of data**
 - L is the latency of the system
 - B is the block size

Reusing Data in the Memory Hierarchy

- **Spatial reuse:** using more than one word in a multi-word line
—using multiple words in a cache line
- **Temporal reuse:** using a word repeatedly
—accessing the same word in a cache line more than once
- Applies at every level of the memory hierarchy
—e.g. TLB
 - spatial reuse: access multiple cache lines in a page
 - temporal reuse: access data on the same page repeatedly

Experimental Study of Memory (membench)

Microbenchmark for memory system performance



for array A of length L from 4KB to 8MB by 2x

for stride s from 4 Bytes (1 word) to L/2 by 2x

time the following loop

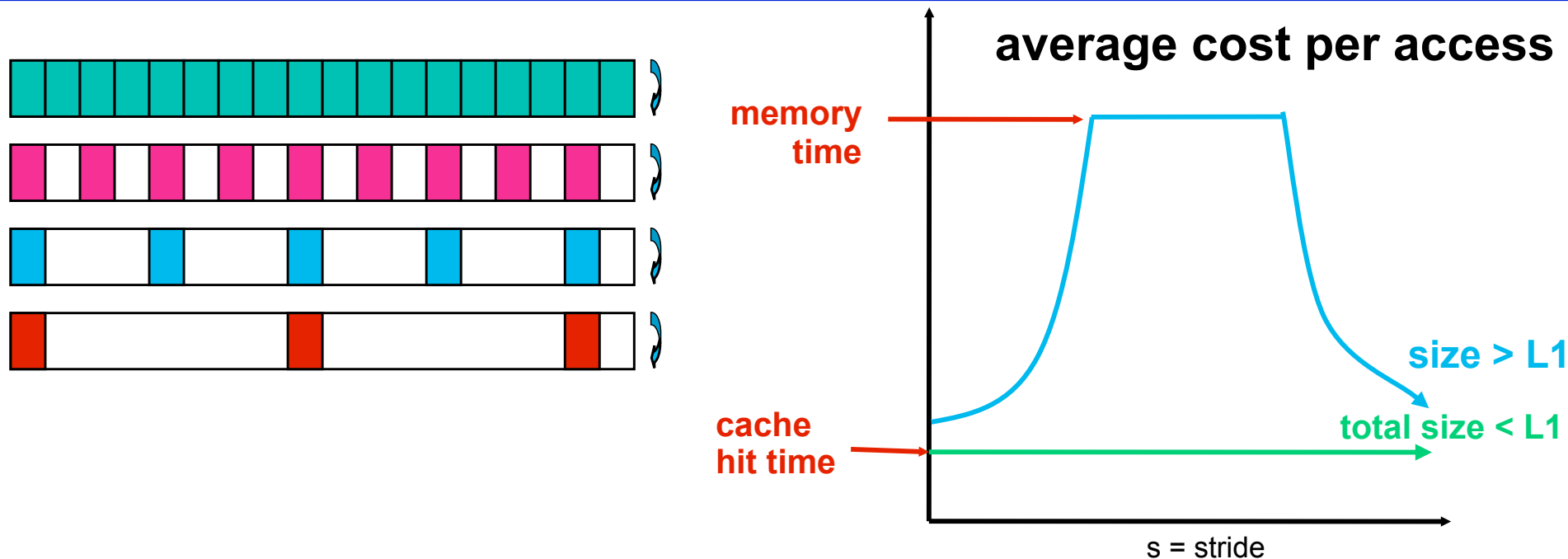
(repeat many times and average)

for i from 0 to L by s

load A[i] from memory (4 Bytes)

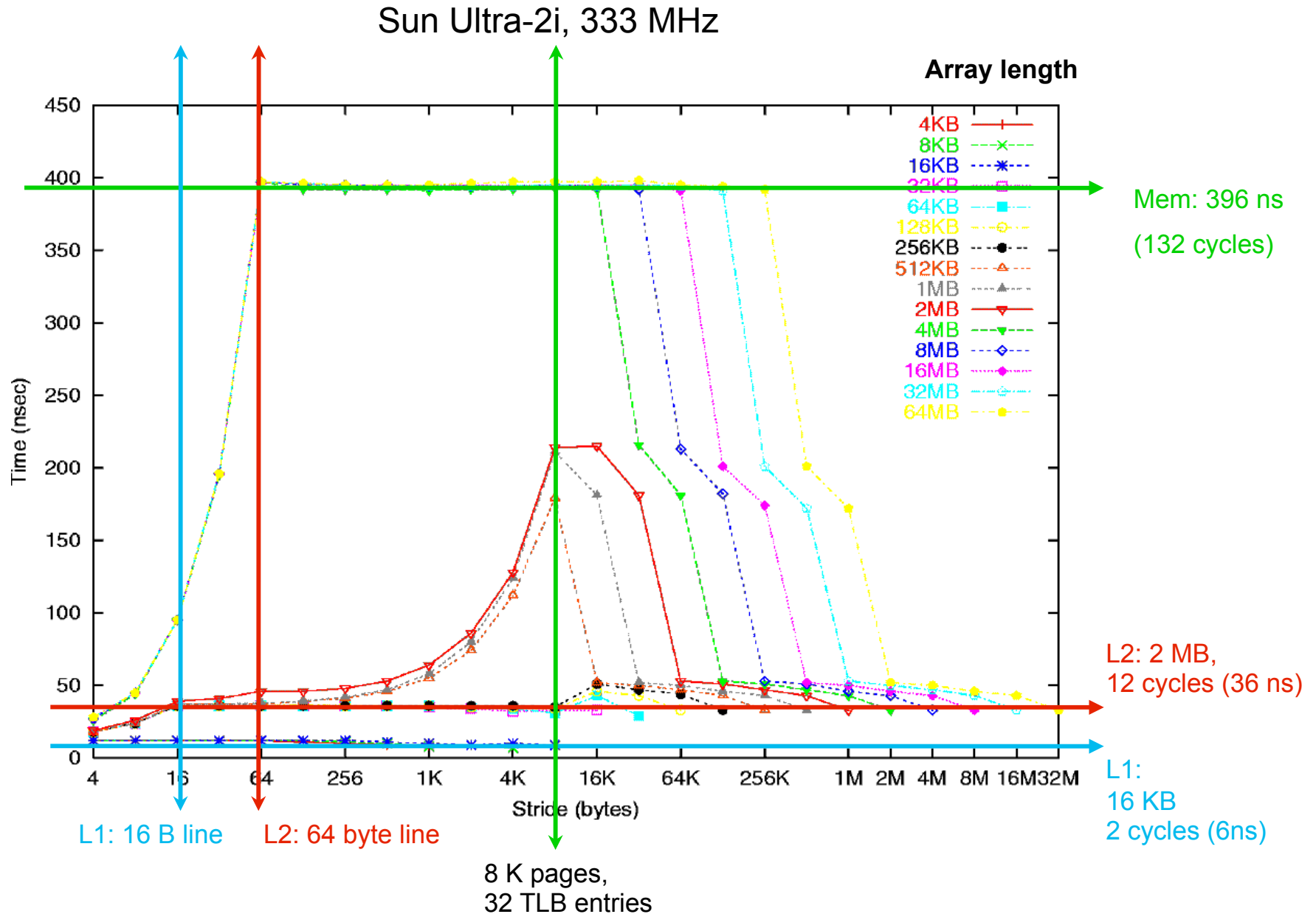
1 experiment

Membench: What to Expect



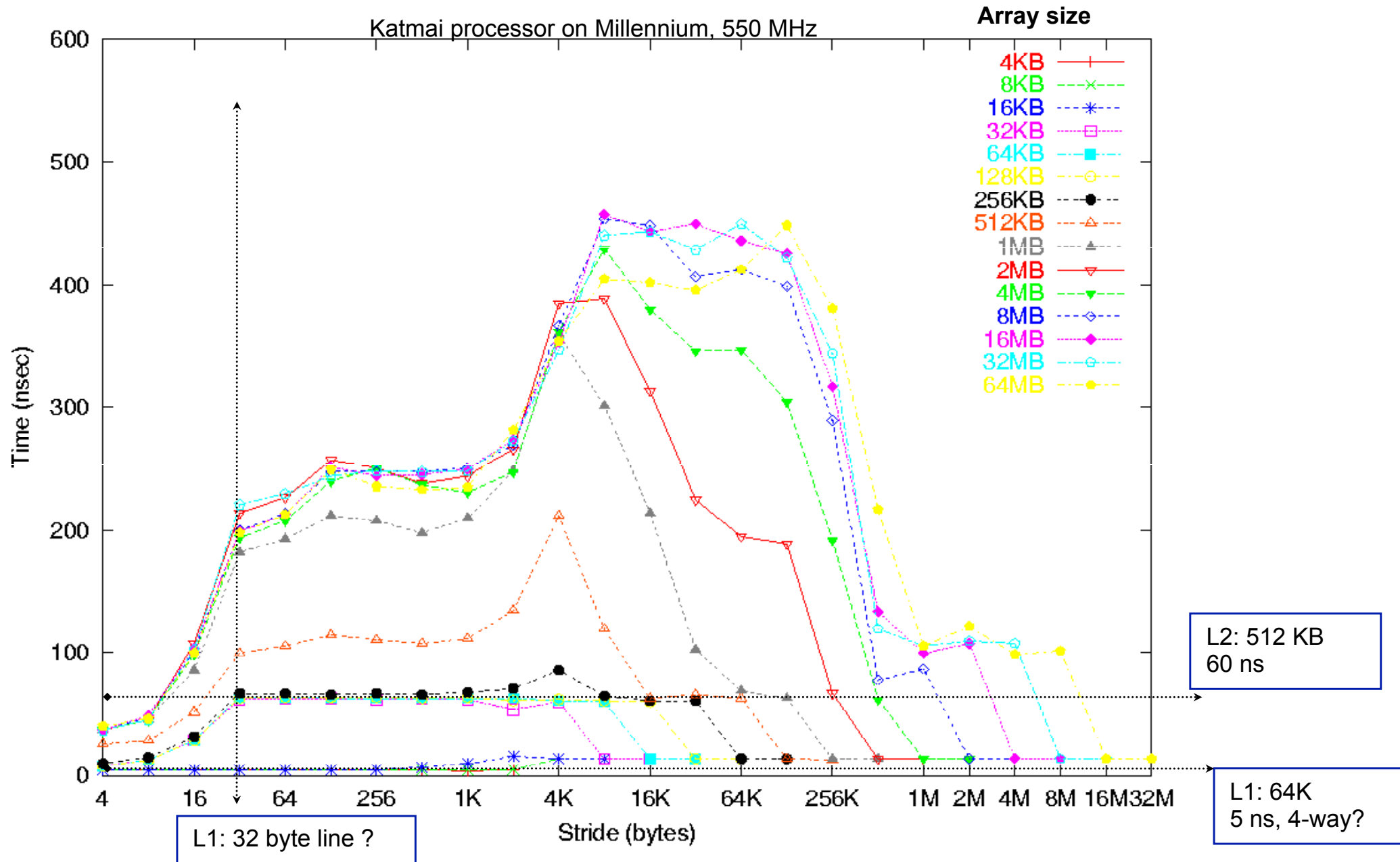
- Consider the average cost per load
 - plot one line for each array length, time vs. stride
 - unit stride is best: if cache line holds 4 words, only $\frac{1}{4}$ miss
 - if array is smaller than a cache, all accesses will hit after first run
 - time for first run is negligible with enough repetitions
 - upper right figure assumes only one level of cache
 - performance profile is more complicated on modern systems

Memory Hierarchy on a Sun Ultra-2i



See www.cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps for details

Memory Hierarchy on a Pentium III



Memory Bandwidth in Practice

What matters for application performance is “balance” between sustainable memory bandwidth and peak double-precision floating-point performance.

Systems at TACC

—Ranger (4-socket quad-core AMD “Barcelona”)

- bandwidth = 7.5 GB/s (2.19 GW/s, 8-Byte Words) per node
- peak FP rate = $2.3 \text{ GHz} * 4 \text{ FP Ops/Hz/core} * 4 \text{ cores/socket} * 4 \text{ sockets} = 147.2 \text{ GFLOPS/node}$
- ratio = 67 FLOPS/Word

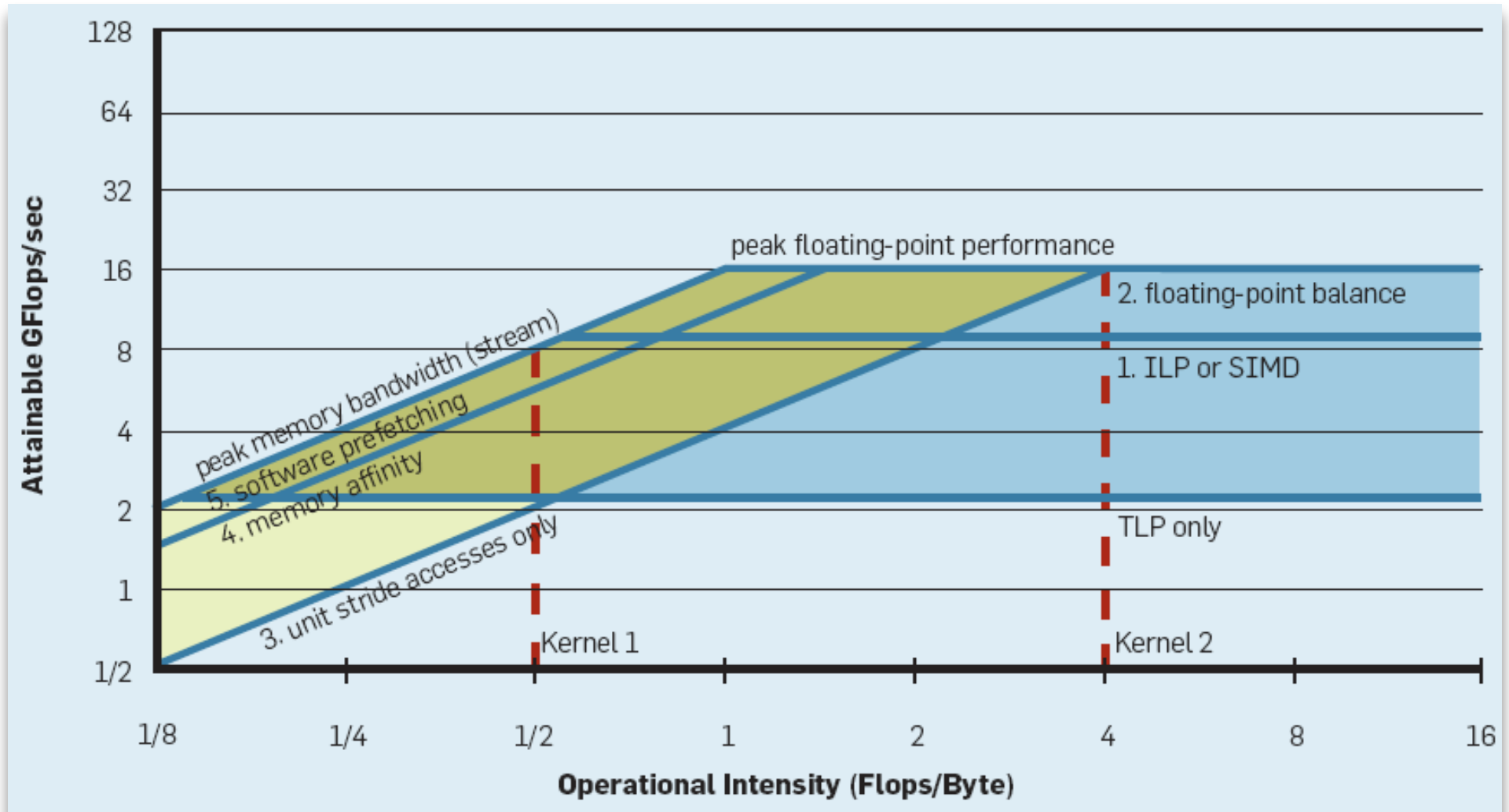
—Lonestar (2-socket 6-core Intel “Westmere”)

- bandwidth = 41 GB/s (5.125 GW/s) per node
- peak FP rate = $3.33 \text{ GHz} * 4 \text{ Ops/Hz/core} * 6 \text{ cores/socket} * 2 \text{ sockets} = 160 \text{ GFLOPS/node}$
- ratio = 31 FLOPS/Word

—Stampede (2-socket 8-core Intel “Sandy Bridge” processors)

- bandwidth = 78 GB/s (9.75 GW/s) per node
- peak FP rate = $2.7 \text{ GHz} * 8 \text{ FP Ops/Hz} * 8 \text{ cores/socket} * 2 \text{ sockets} = 345.6 \text{ GFLOPS per node}$
- ratio = 35 FLOPS/Word

Understanding Performance Limitations



Williams, Waterman, Patterson; CACM April 2009

Memory System Performance: Summary

- Exploiting spatial and temporal locality is critical for
 - amortizing memory latency
 - increasing effective memory bandwidth
- Ratio # operations / # memory accesses
 - good indicator of anticipated tolerance to memory bandwidth
- Memory layout and computation organization significantly affect spatial and temporal locality

Multithreading for Latency Hiding

- A thread is a single stream of control in the flow of a program.
- We illustrate threads with a dense matrix vector multiply

```
for (i = 0; i < n; i++)  
    c[i] = dot_product(get_row(a, i), b);
```

- Each dot-product is independent of others
—thus, can execute concurrently
- Can rewrite the above code segment using threads

```
for (i = 0; i < n; i++)  
    c[i] = create_thread(dot_product, get_row(a, i), b);
```


Multithreading for Latency Hiding (contd)

- **Consider how the code executes**
 - first thread accesses a pair of vector elements and waits for them
 - second thread can access two other vector elements in the next cycle
 - ...
- **After L units of time**
 - (L is the latency of the memory system)
 - first thread gets its data from memory and performs its madd
- **Next cycle**
 - data items for the next function instance arrive
- ...
- **Every clock cycle, we can perform a computation**

Multithreading for Latency Hiding (contd)

- Previous example makes two hardware assumptions
 - memory system can service multiple outstanding requests
 - processor is capable of switching threads at every cycle
- Also requires program to have explicit threaded concurrency
- Machines such as the Sun T2000 (Niagara-2) and the Cray Threadstorm rely on multithreaded processors
 - can switch the context of execution in every cycle
 - are able to hide latency effectively
- Sun T2000, 64-bit SPARC v9 processor @1200MHz
 - organization: 8 cores, 4 strands per core, 8KB Data cache and 16KB Instruction cache per core, L2 cache: unified 12-way 3MB, RAM: 32GB
- Cray Threadstorm: 128 threads

Prefetching for Latency Hiding

- Misses on loads cause programs to stall; why not load data before it is needed?
 - by the time it is actually needed, it will be there!
- Drawback: need space to store early loads
 - may overwrite other necessary data in cache
 - if early loads are overwritten, we are little worse than before!
- Prefetching support
 - software only, e.g. Itanium2
 - hardware and software, modern Intel, AMD, ...
- Hardware prefetching requires
 - predictable access pattern
 - limited number of independent streams

Tradeoffs in Multithreading and Prefetching

- **Multithreaded systems**
 - bandwidth requirements
 - may increase very significantly because of reduced cache/ thread
 - can become bandwidth bound instead of latency bound
- **Multithreading and prefetching**
 - only address latency
 - may often exacerbate bandwidth needs
 - have significantly larger data footprint; need hardware for that

References

- Adapted from slides “Parallel Programming Platforms” by Ananth Grama accompanying course textbook
- Vivek Sarkar (Rice), COMP 422 slides from Spring 2008
- Jack Dongarra (U. Tenn.), CS 594 slides from Spring 2008, <http://www.cs.utk.edu/%7Edongarra/WEB-PAGES/cs594-2008.htm>
- Kathy Yelick (UC Berkeley), CS 267 slides from Spring 2007, http://www.eecs.berkeley.edu/~yelick/cs267_sp07/lectures
- Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam and P. Sadayappan. Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures. In ETAPS Intl. Conf. on Compiler Construction (CC'2011), Springer Verlag, Saarbrücken, Germany, March 2011.