Parallel Computing Platforms: Control Structures and Memory Hierarchy

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu

Topics for Today

- **SIMD, MIMD, SIMT 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

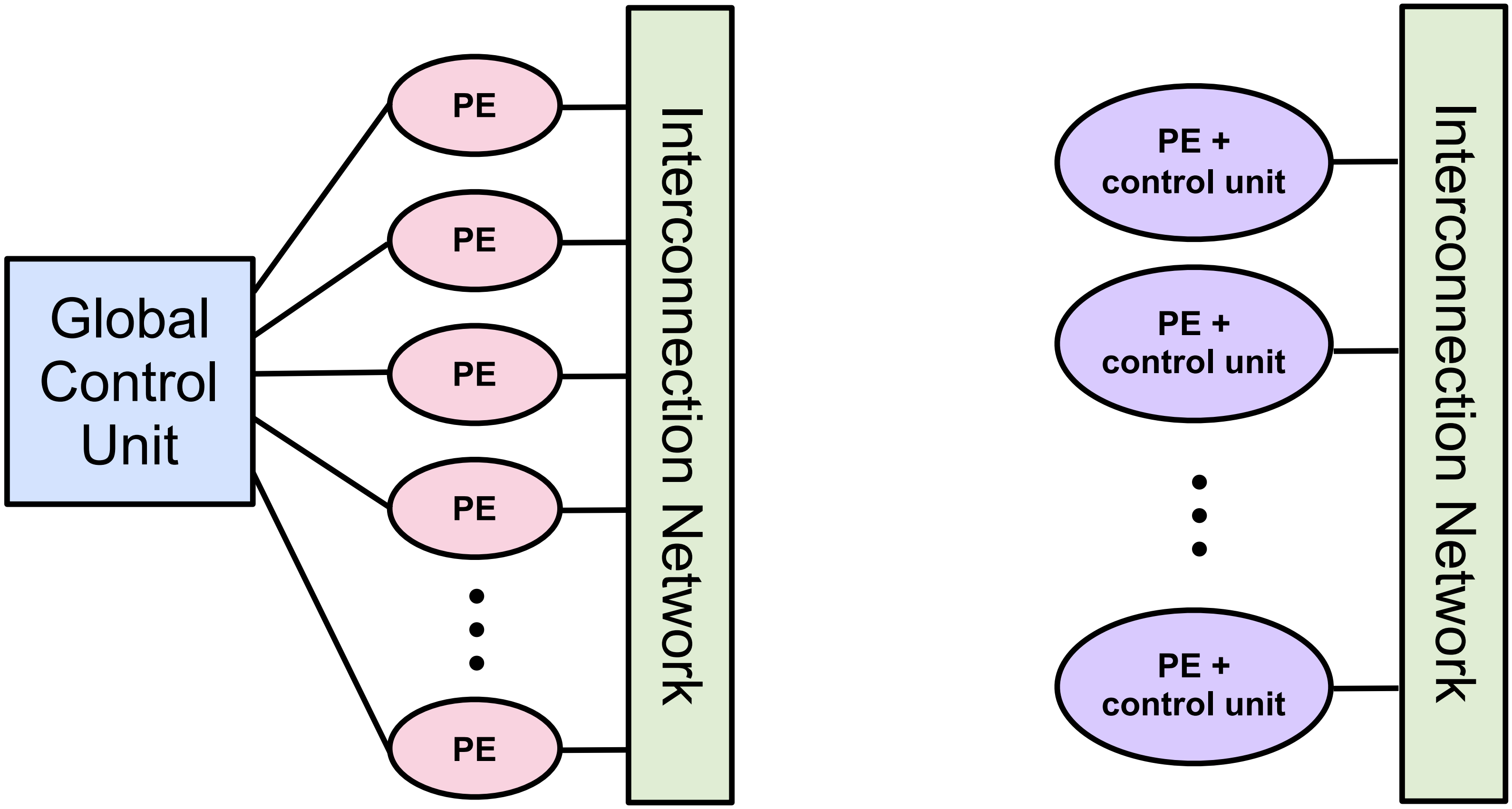
- **Processor control structure alternatives**
  - work independently
  - operate under the centralized control of a single control unit
- **MIMD**
  - Multiple Instruction streams
    - each hardware thread has its own control unit
    - each hardware thread can execute different instructions
  - Multiple 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

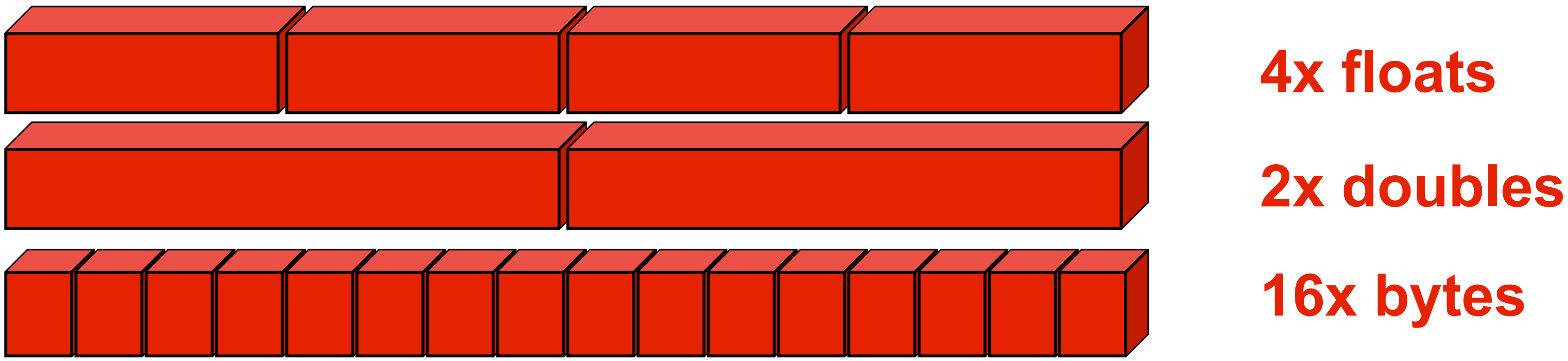
PE = Processing Element

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

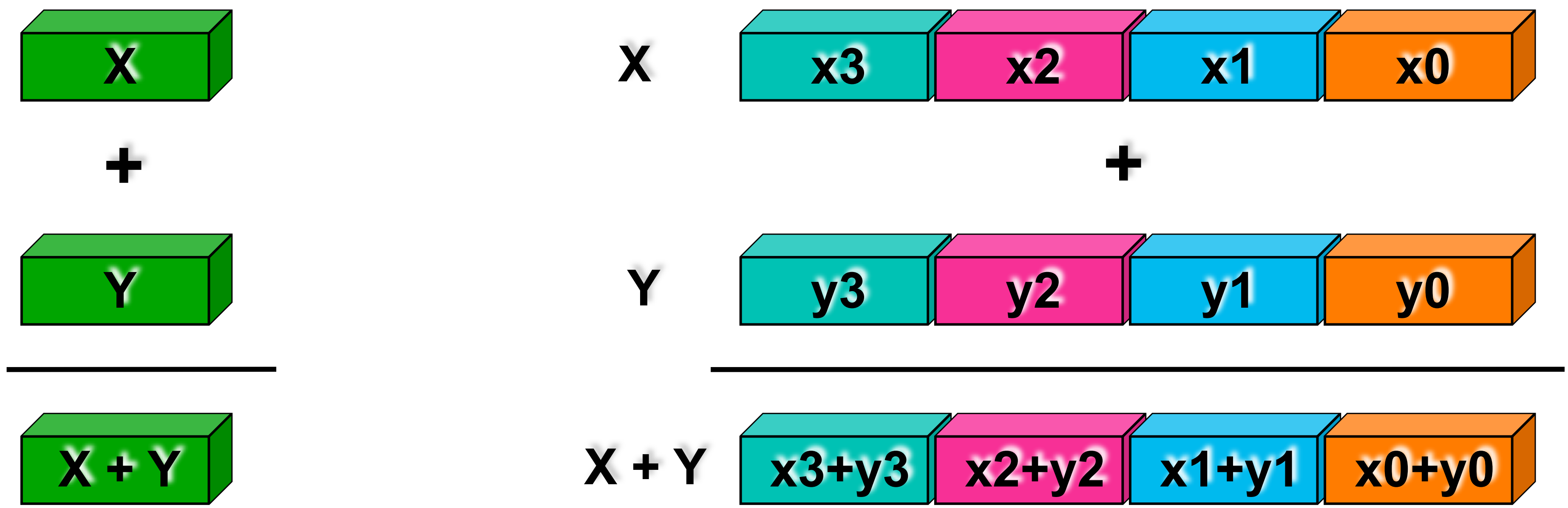
- Data types: anything that fits into 16 bytes, e.g.,



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



Slide Credit: Alex Klimovitski & Dean Macri, Intel Corporation

Executing a Conditional on a SIMD Processor

conditional statement

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

initial values

	Processor 0	Processor 1	Processor 2	Processor 3
A	0	4	2	0
B	5	8	2	7
C	0	0	0	0

execute "then" branch

	Processor 0	Processor 1	Processor 2	Processor 3
A	0	4	1	0
B	5	8	2	7
C	5	0	0	7

execute "else" branch

	Processor 0	Processor 1	Processor 2	Processor 3
A	0	4	2	0
B	5	8	2	7
C	5	2	1	7

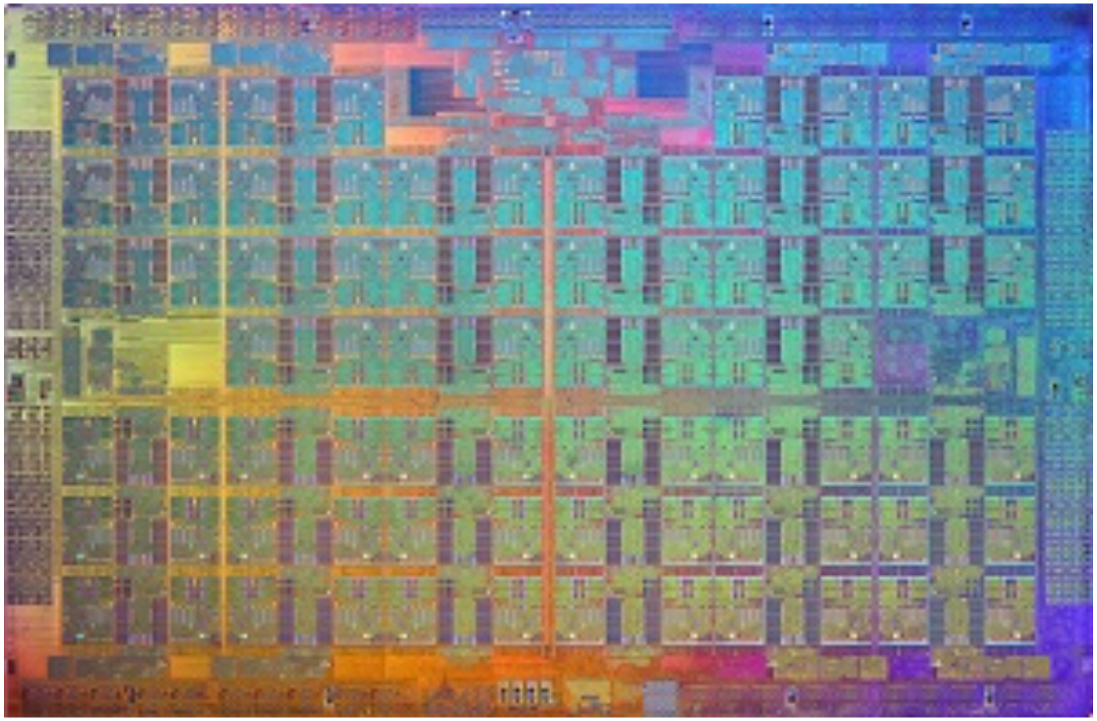
SIMD Examples

- 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 functional units or co-processors
  - —vector units
    - – 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
    - – VSX - Vector-Scalar Extensions
      - 64 128-bit vector registers in IBM Power processors
        - all can be used for vector-scalar floating point operations
        - 32 of these registers can be used as 8/16/32/64/128-bit quantities
  - —co-processors
    - – ClearSpeed CSX700 array processor (control PE + array of 96 PEs)
    - – NVIDIA Volta V100 GPGPU

Intel Knight's Landing (includes SIMD)

- > 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

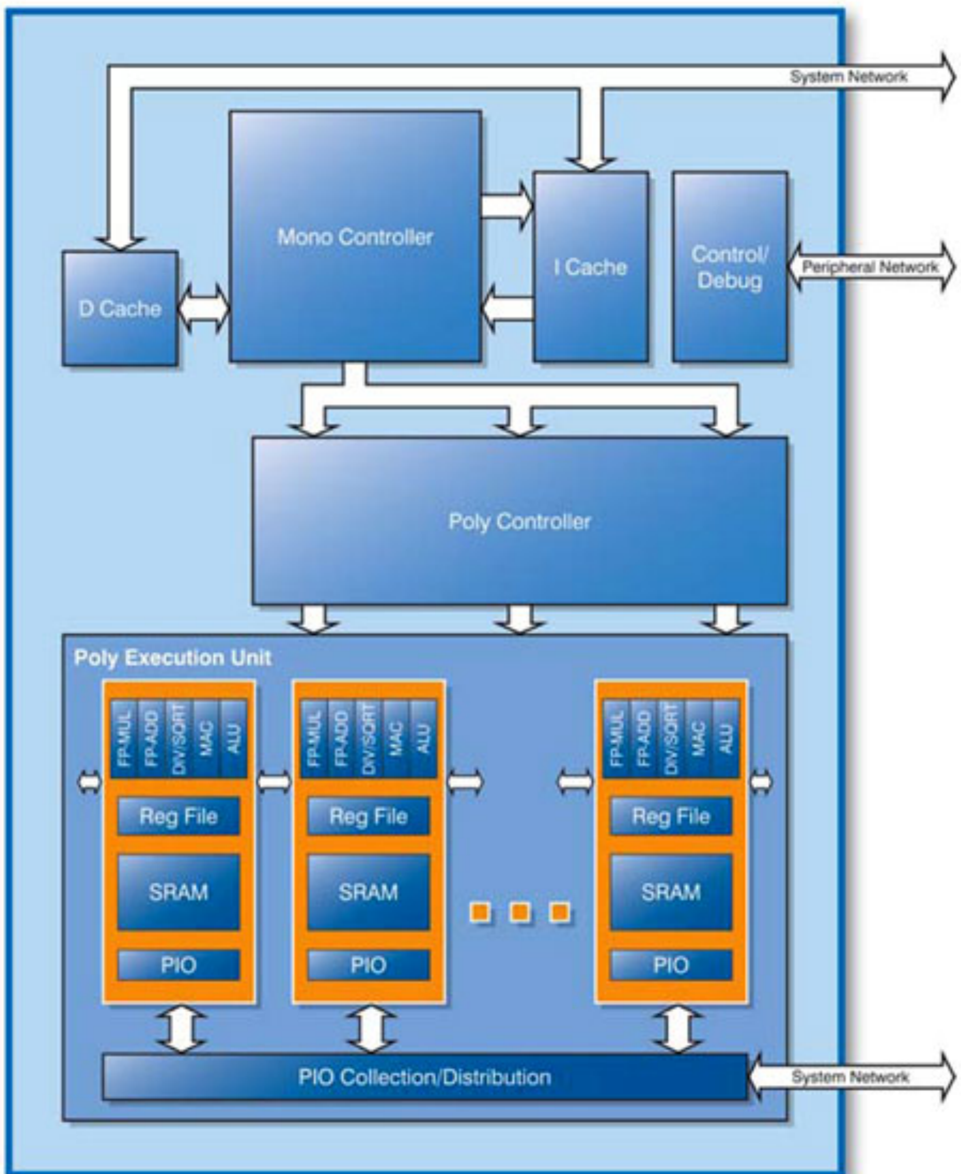
2nd Generation Xeon Phi



http://ark.intel.com/products/95831/Intel-Xeon-Phi-Processor-7290F-16GB-1_50-GHz-72-core

SIMD: ClearSpeed MTAP Co-processor

MTAP processor



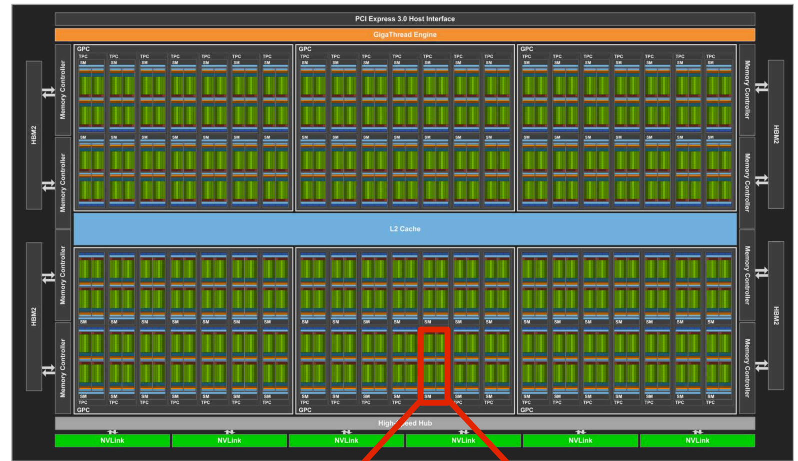
- **Features**
  - —hardware multi-threading
  - —asynchronous, overlapped I/O
  - —extensible instruction set
- **SIMD core**
  - —poly controller
  - —poly execution unit
    - – array of 192 PEs
    - – 64- and 32-bit floating point
    - – 250 MHz (key to low power)
    - – 96 GFLOP, <15 Watts

(CSX700 released June 2008 company delisted in 2009)

csx700_product_brief.pdf

NVIDIA VOLTA V100 (SIMT)

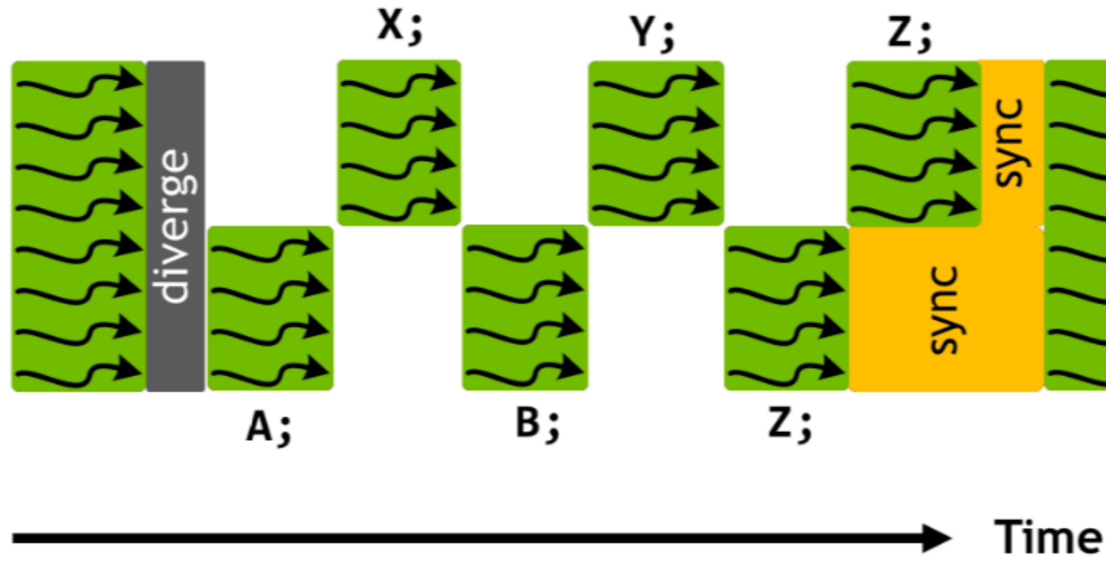
- **21.1B transistors**
- **84 Streaming Multiprocessors (SMs)**
- **Each SM**
  - —64 FP32 cores
  - —64 INT32 cores
  - —32 FP64 cores
  - —8 tensor cores (64 FP16 FMA each/cycle)
  - —4 texture units
  - —4 warp schedulers
    - – 32-thread groups (warp)
    - – 4 warps issue and execute concurrently
- **7.8 TF DP; 125 Tensor TF**



http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

SIMT Thread Scheduling on Volta

```
if (threadIdx.x < 4) {
     A;
     B;
} else {
     X;
     Y;
}
Z;
__syncwarp()
```



Independent thread scheduling enables threads in a warp to execute independently - a key to starvation freedom when threads synchronize

```
__device__ void insert_after(Node *a, Node *b)
{
    Node *c;
    lock(a); lock(a->next);
    c = a->next;

    a->next = b;
    b->prev = a;

    b->next = c;
    c->prev = b;

    unlock(c); unlock(a);
}
```

NVIDIA TESLA V100 GPU ARCHITECTURE
White Paper WP-08608-001_v1.1, August 2017

SIMT Thread Scheduling on Volta

```
__device__ void insert_after(Node *a, Node *b)
{
    Node *c;
    lock(a); lock(a->next);
    c = a->next;

    a->next = b;
    b->prev = a;

    b->next = c;
    c->prev = b;

    unlock(c); unlock(a);
}
```

Short Vectors: The Good and Bad

```
      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];
      }
```

Performance:	AMD Phenom	1.2 GFlop/s
	Core2	3.5 GFlop/s
	Core i7	4.1 GFlop/s

(a) Stencil code

```
      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];
      }
```

Performance:	AMD Phenom	**1.9** GFlop/s
	Core2	**6.0** GFlop/s
	Core i7	**6.7** GFlop/s

(b) Non-Stencil code

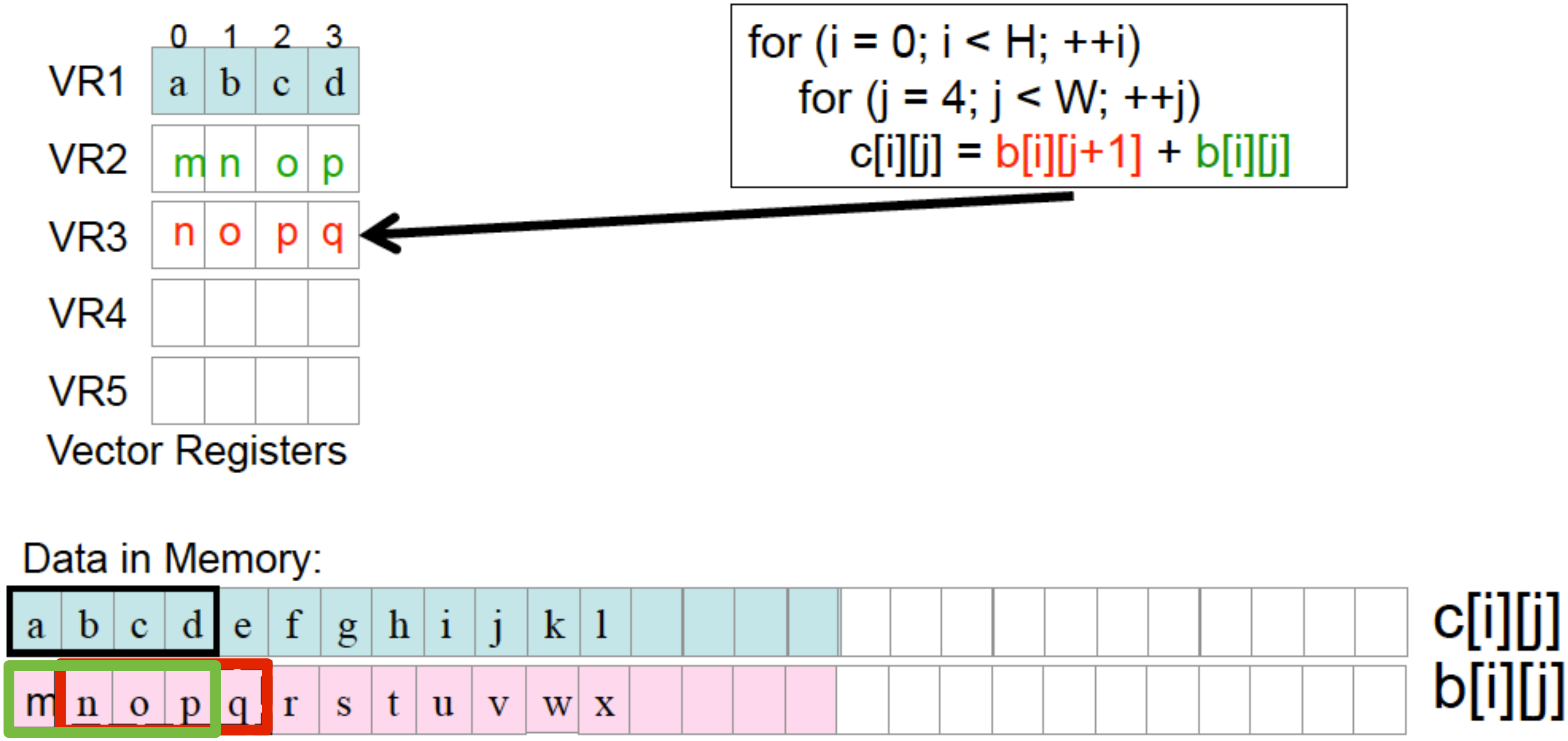
The stencil code (a) has much lower performance than the non-stencil code (b) despite accessing 50% fewer data elements

Figure credit: P. Sadayappan. See Henretty et al. [CC'11]

The Subtlety of Using Short Vectors

- **Consider the following:**

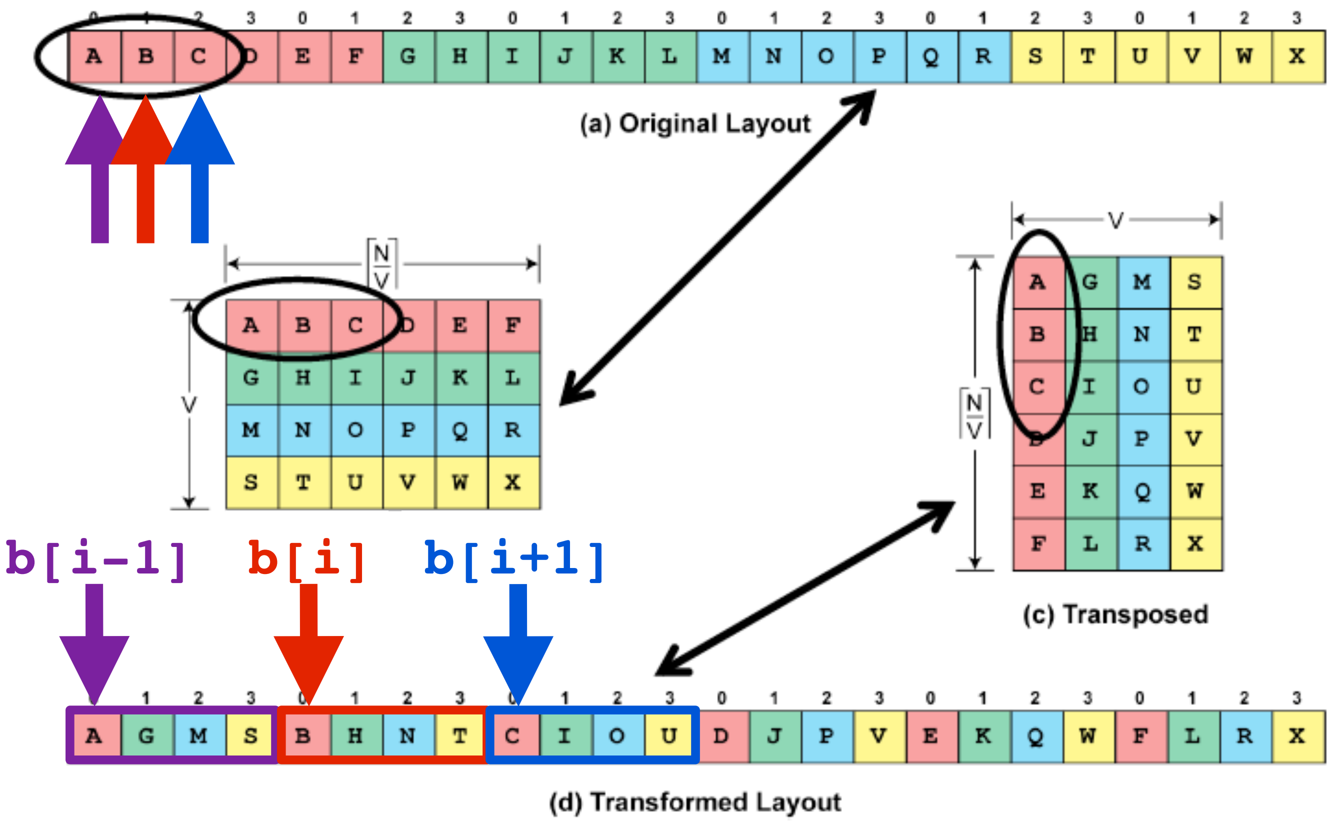
```
for (i = 0; i < H; ++i)
    for (j = 4; j < W; ++j)
        c[i][j] = b[i][j+1] + b[i][j]
```

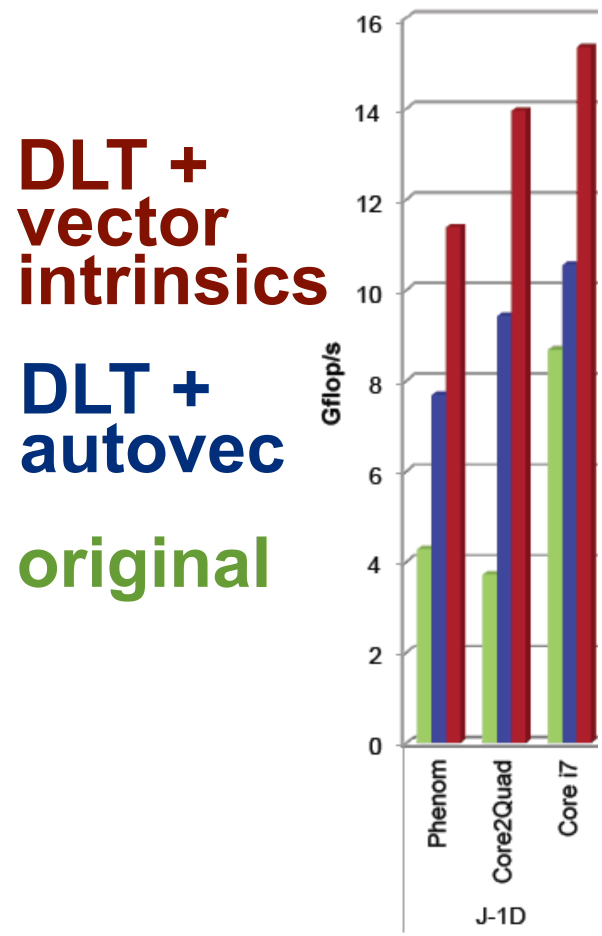


- **Stream alignment conflict between b[i][j+1] and c[i][j]**

Figure credit: P. Sadayappan. See Henretty et al. [CC'11]

Dimension-lifted Transformation (DLT)





(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]

Figure credit: P. Sadayappan. See Henretty et al. [CC'11]

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)
  - distributed memory
    - clusters (e.g., nots.rice.edu, davinci.rice.edu)
    - Cray XC, IBM Blue Gene, Power9+NVIDIA Volta
- SPMD programming paradigm
  - Single Program, Multiple Data streams
    - same program on different PEs, behavior conditional on thread id

SIMD, MIMD, SIMT

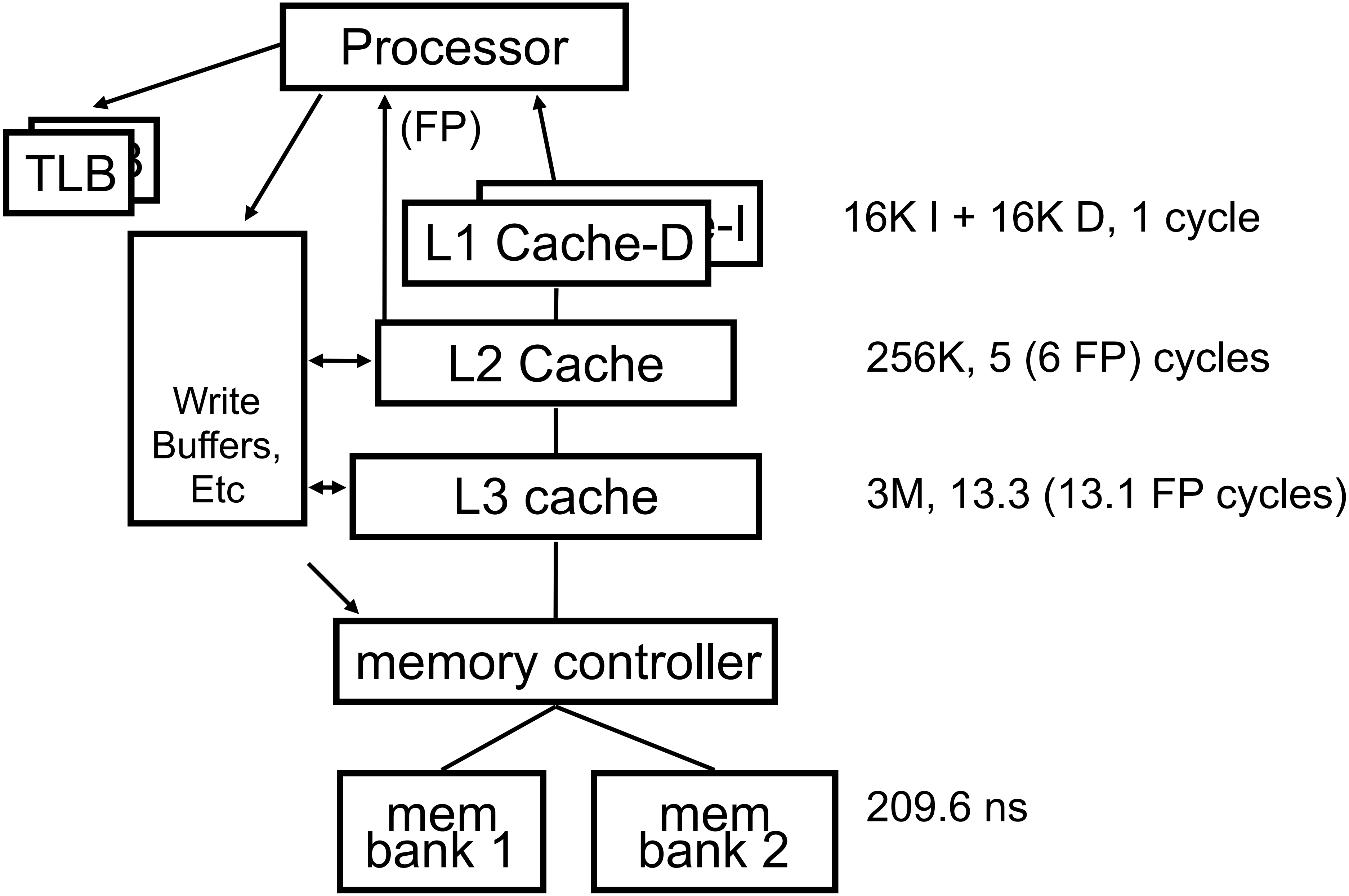
- **SIMD platforms**
  - —special purpose: not well-suited for all applications
  - —custom designed with long design cycles
  - —less hardware: single control unit
  - —need less memory: only 1 copy of program
  - —today: SIMD common only for vector units
- **MIMD platforms**
  - —suitable for broad range of applications
  - —inexpensive: off-the-shelf components + short design cycle
  - —need more memory: program and OS on each processor
- **SIMT**
  - —GPUs, e.g., NVIDIA VOLTA

Data Movement and Communication

- **Latency**: How long does a single operation take?
  - measured in nanoseconds
- **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)



http://www.devx.com/Intel/Article/20521

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

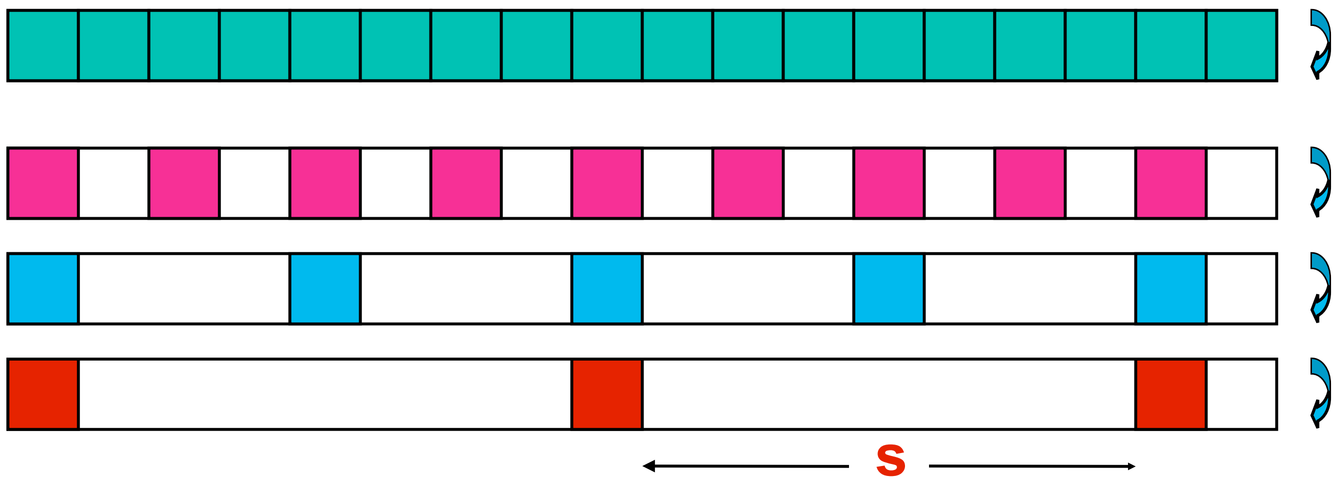
Reusing Data in the Memory Hierarchy

- **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)

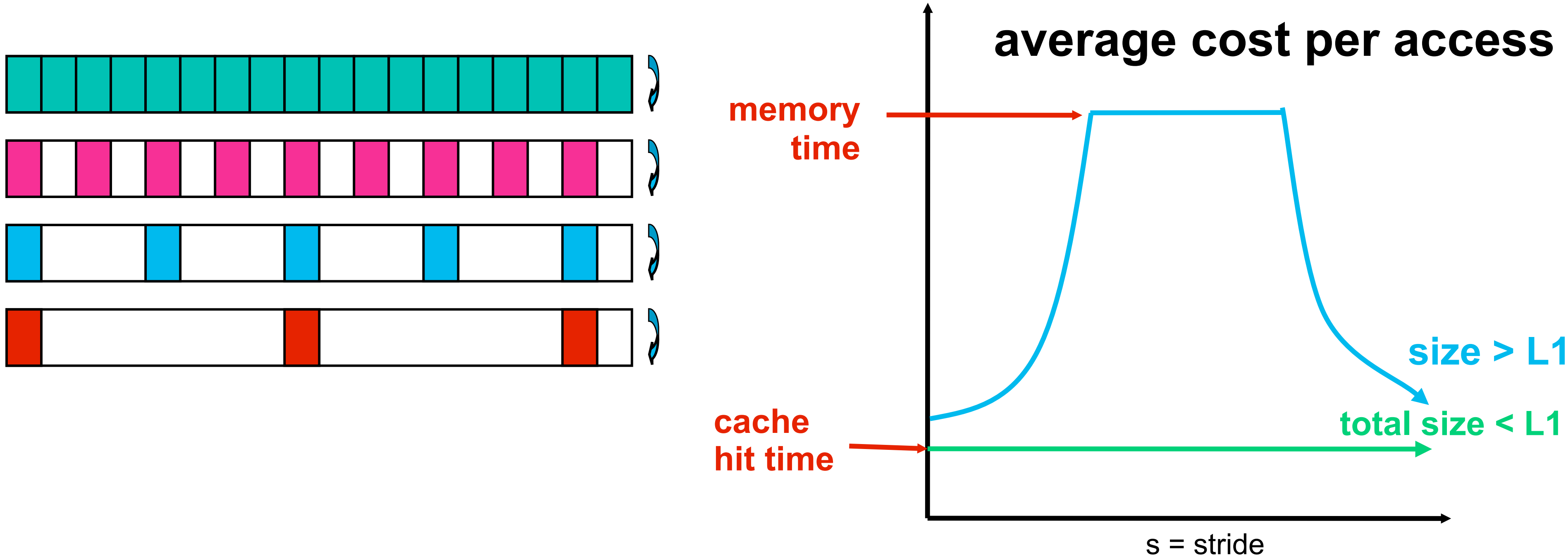
Microbenchmark for memory system performance



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)

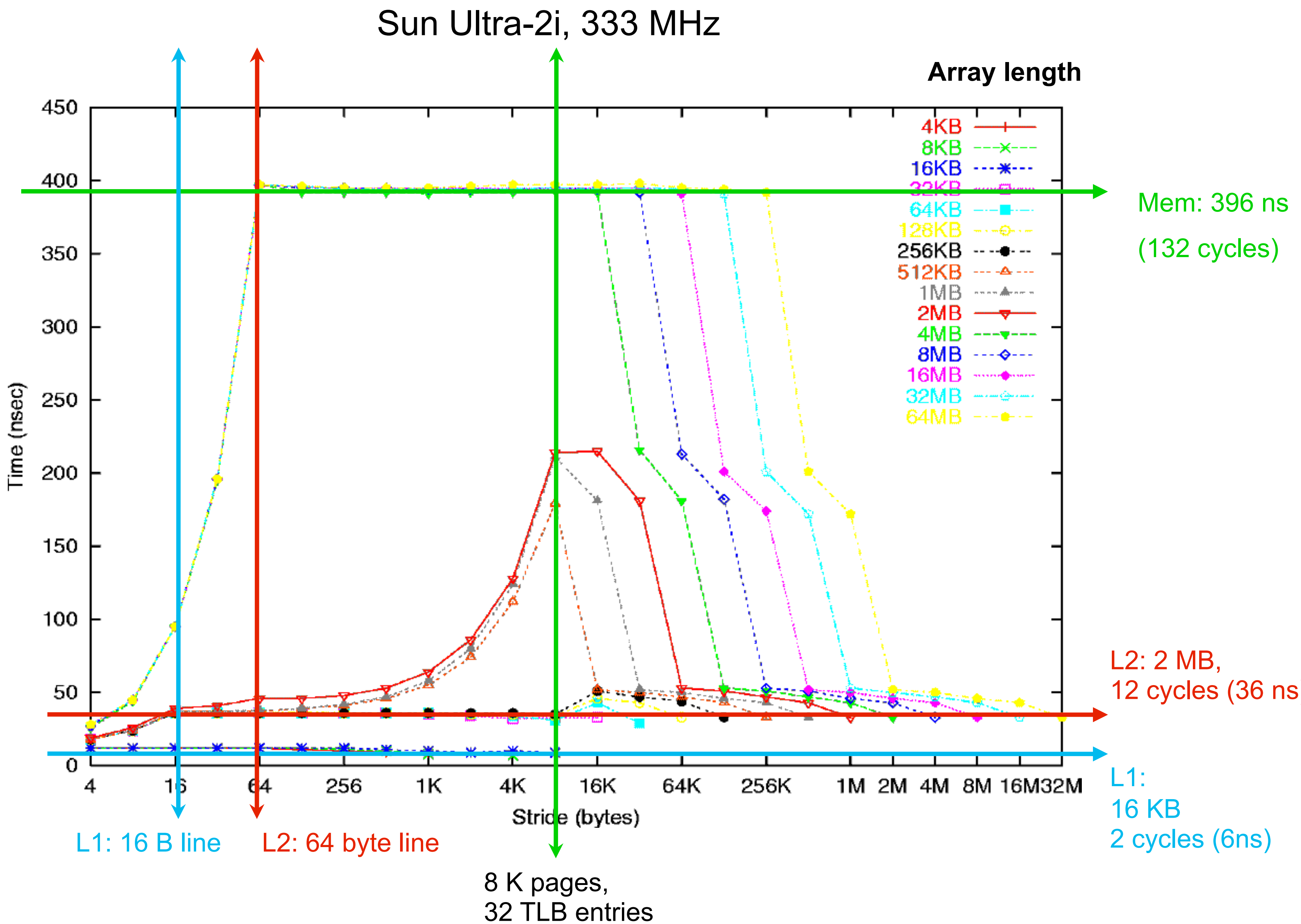
1 experiment

Membench: What to Expect



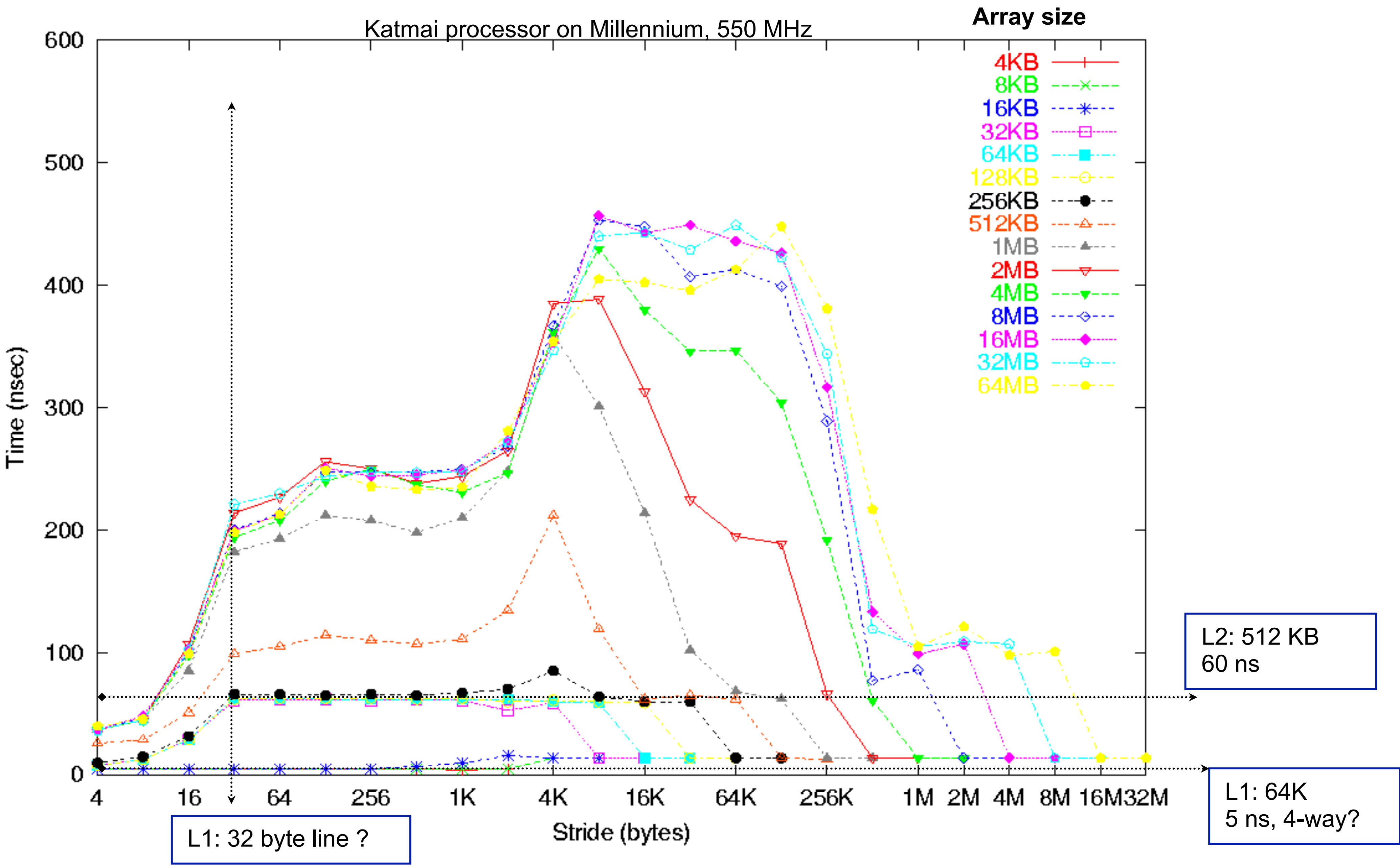
- **Consider the average cost per load**
  - —plot one line for each array length, time vs. stride
  - —unit stride is best: if cache line holds 4 words, only ¼ miss
  - —if array is smaller than a cache, all accesses will hit after first run
    - – time for first run is negligible with enough repetitions
  - —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

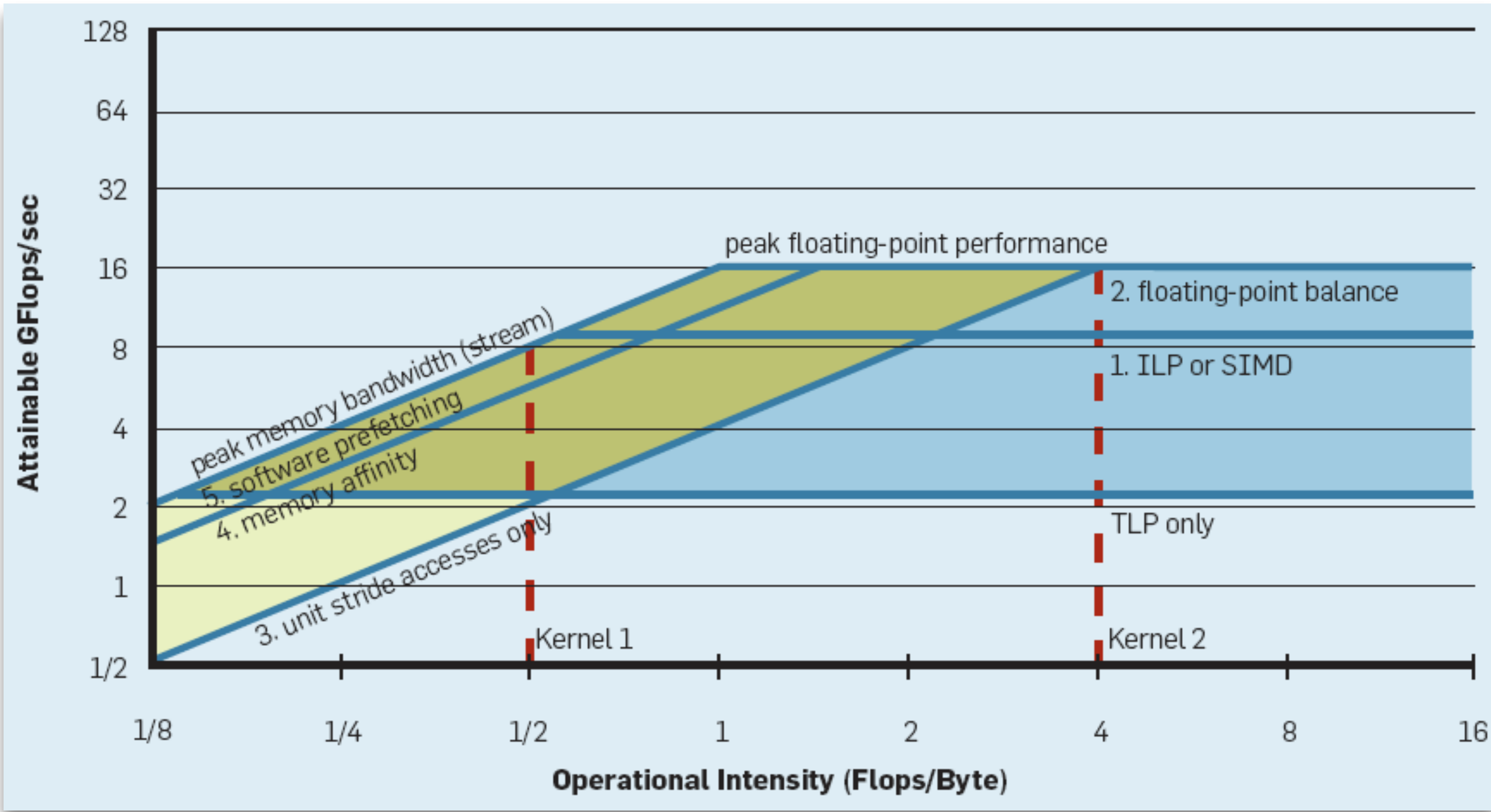
What matters for application performance is "balance" between sustainable memory bandwidth and peak double-precision floating-point performance.

Analysis of some prior systems at Texas Advanced Computing Center

- **—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 GHz * 4 FP Ops/Hz/core * 4 cores/socket * 4 sockets = 147.2 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 GHz * 4 Ops/Hz/core * 6 cores/socket * 2 sockets = 160 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 GHz * 8 FP Ops/Hz * 8 cores/socket * 2 sockets = 345.6 GFLOPS per node**
  - **ratio = 35 FLOPS/Word**

Credit: John McCalpin, TACC, http://blogs.utexas.edu/jdm4372/2012/11/

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

- 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

```
#pragma omp parallel for
for (i = 0; i < n; i++)
  c[i] = 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, Saarbrucken, Germany, March 2011.**