

MSc Course - High Performance Computing

Single Processor Machines: **Memory Hierarchies, Memory Bandwidth, Processor Features and Locality**

Olaf Schenk

ICS - Institute of Computational Science

USI Lugano, Switzerland

October 02, 2018

Outline

Introduction memory hierarchy

- Cache mappings
- Memory access
- Case study: Matrix-matrix multiplication

Motivation

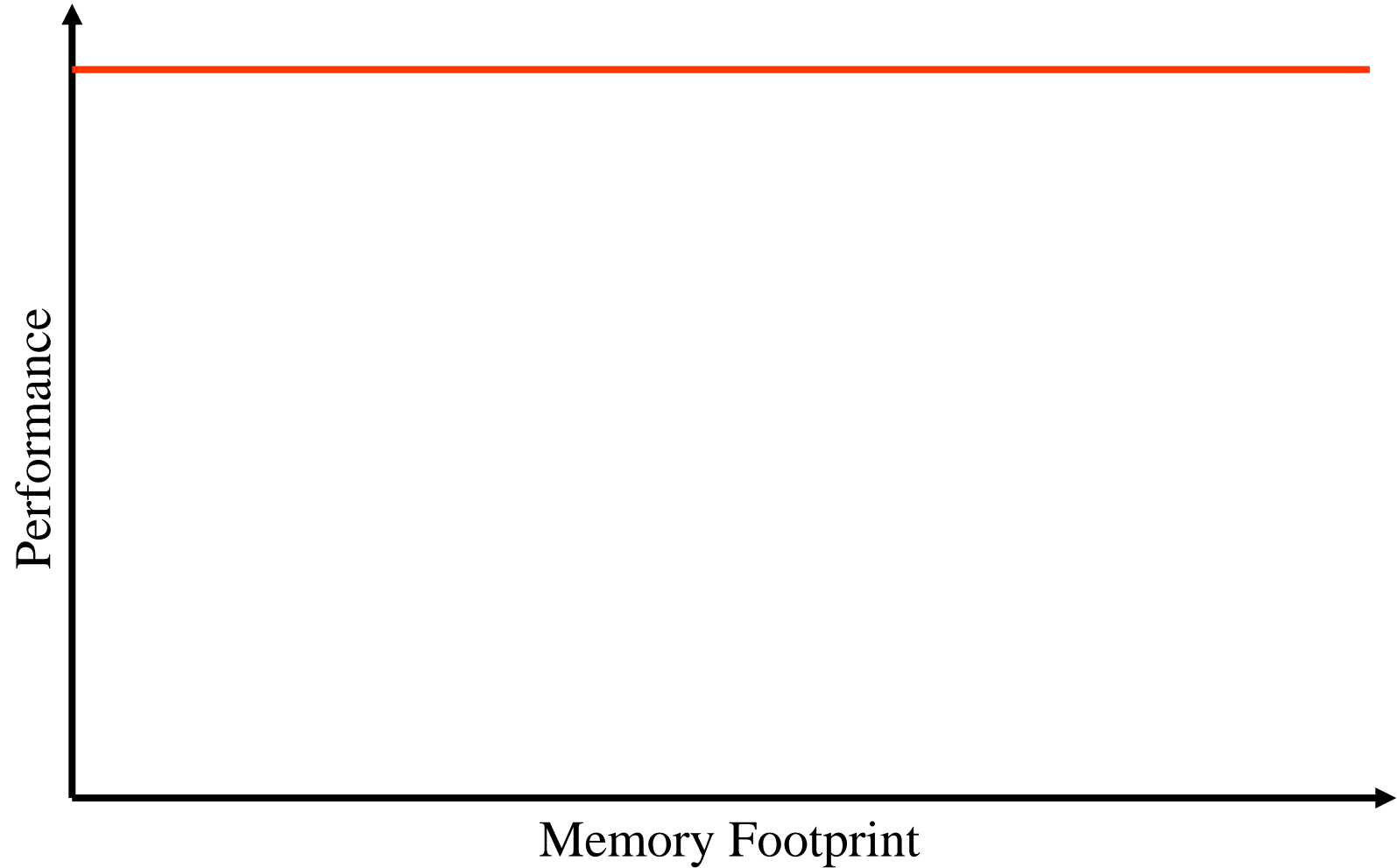
- Most applications run at $< 5\%$ of the “peak” performance of a system
 - **Peak is the maximum the hardware can physically execute**
- Much of this performance is lost on a single processor, i.e., the code running on one processor often runs at only 5% of the processor peak
- Most of the single processor performance loss is in the memory system
 - **Moving data takes much longer than arithmetic and logic**
- To understand this, we need to look under the hood of modern processors
 - **For today, we will look at only a single “core” processor**
 - **These issues will exist on processors within any parallel computer**

Uniprocessors in the Real World

- Real single-core processors have
 - **registers and caches**
 - small amounts of fast memory, store values of recently used or nearby data, different memory ops can have very different costs
 - **parallelism**
 - multiple “functional units” that can run in parallel
 - different orders, instruction mixes have different costs
 - **pipelining**
 - a form of parallelism, like an assembly line in a factory
- Why is this your problem?
 - In theory, compilers and hardware “understand” all this and can optimize your program; in practice they don’ t.
 - They won’ t know about a different algorithm that might be a much better “match” to the problem

***In theory there is no difference between theory and practice.
But in practice there is. -J. van de Snepscheut***

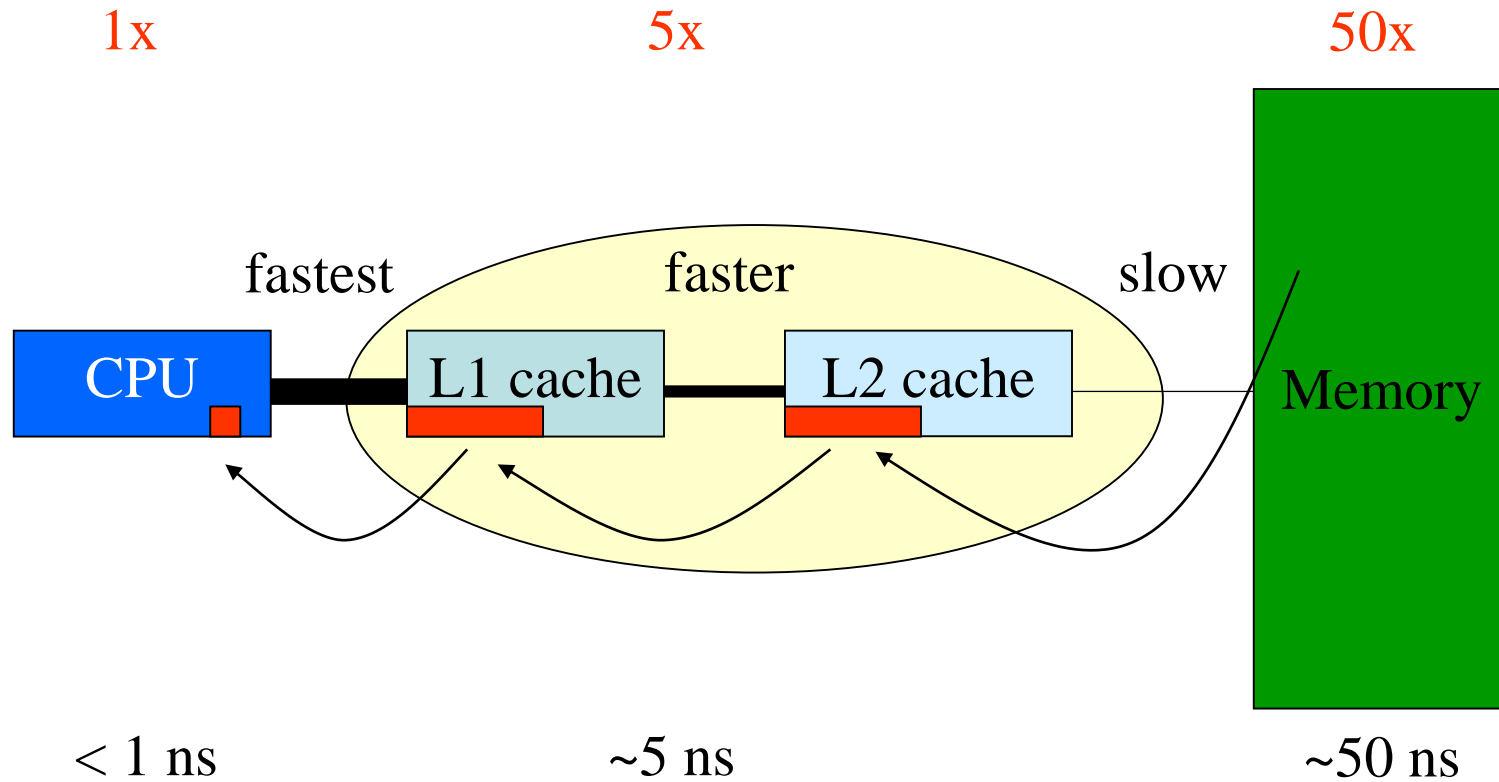
Intuitive Performance Graph



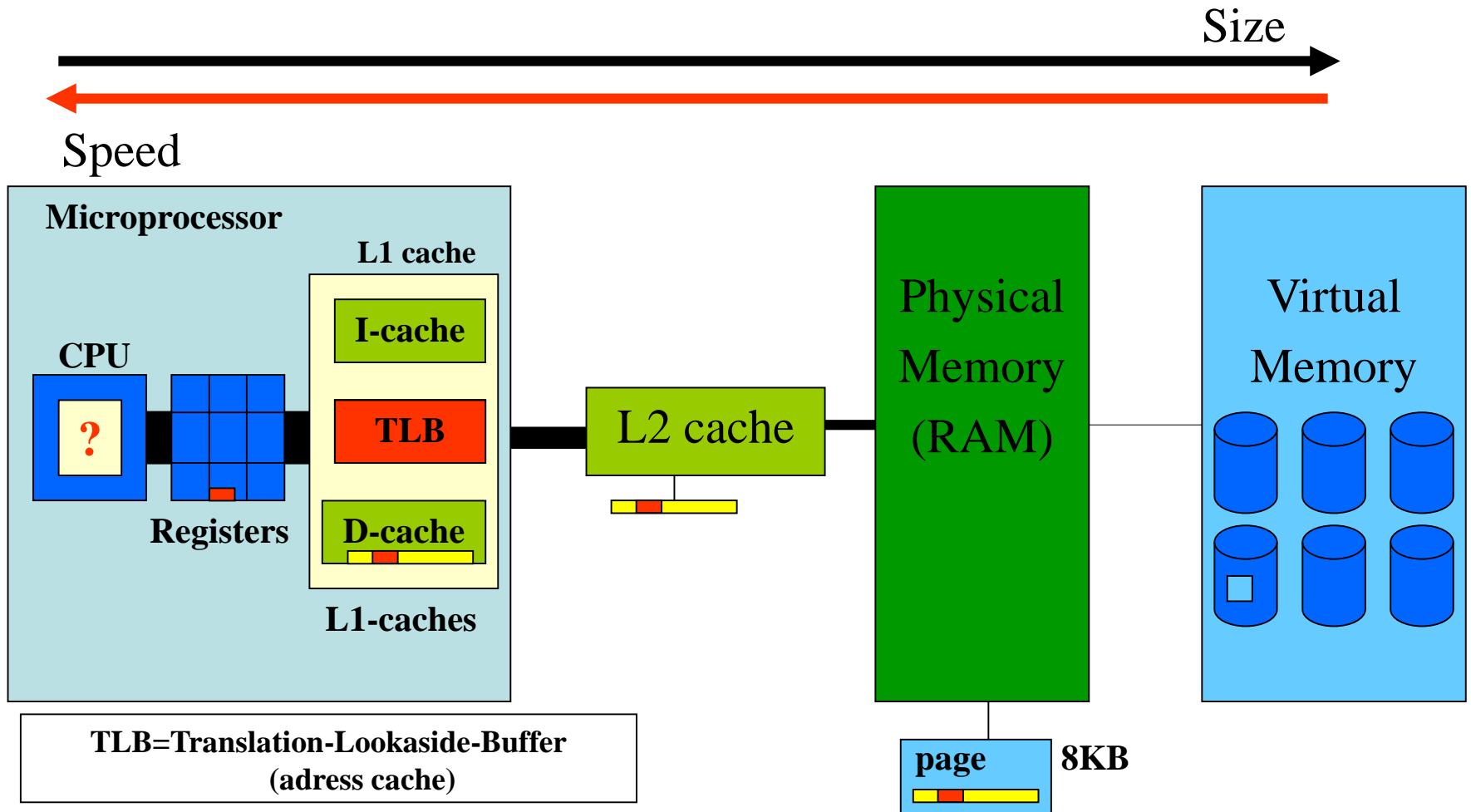
About Memory

- Memory plays a crucial role in performance
- Not accessing memory in the right way will degrade performance on all computer systems
- The extent of the degradation depends on the system
- Knowing more about some of the relevant memory characteristics will help you to write codes such that the problem will be non-existent, or at least minimal

Typical Cache Based System



The Memory Hierarchy



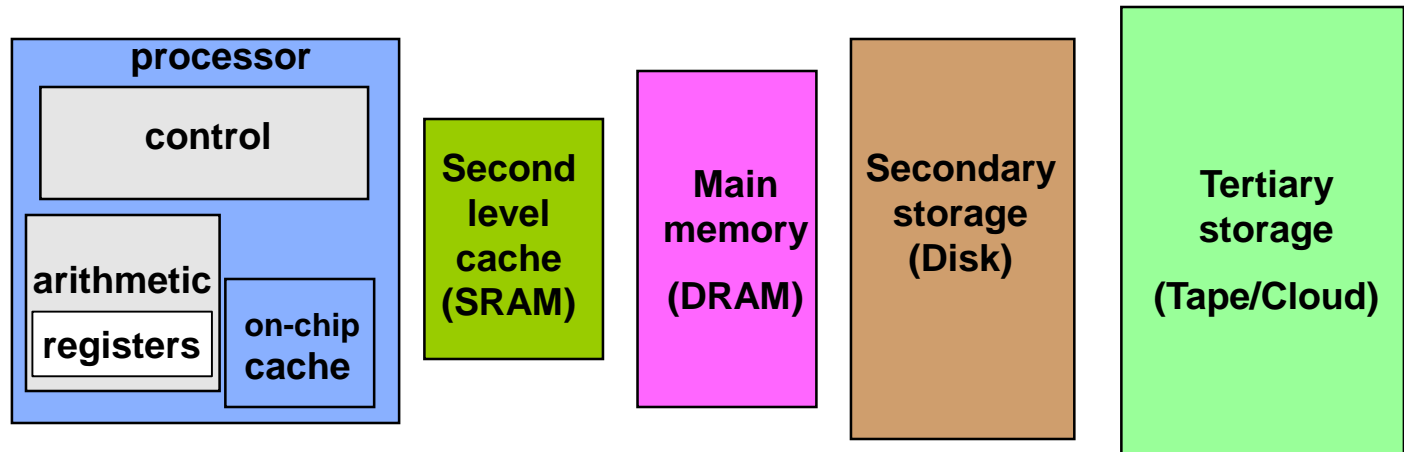
Typical size (e.g. Intel Xeon):

- L2 cache 2 MB (=1024 cache lines)
- Cache line 32×64 Bytes = 2 K

Memory Optimization:
Keep frequently used data close to the processor

The Memory Hierarchy

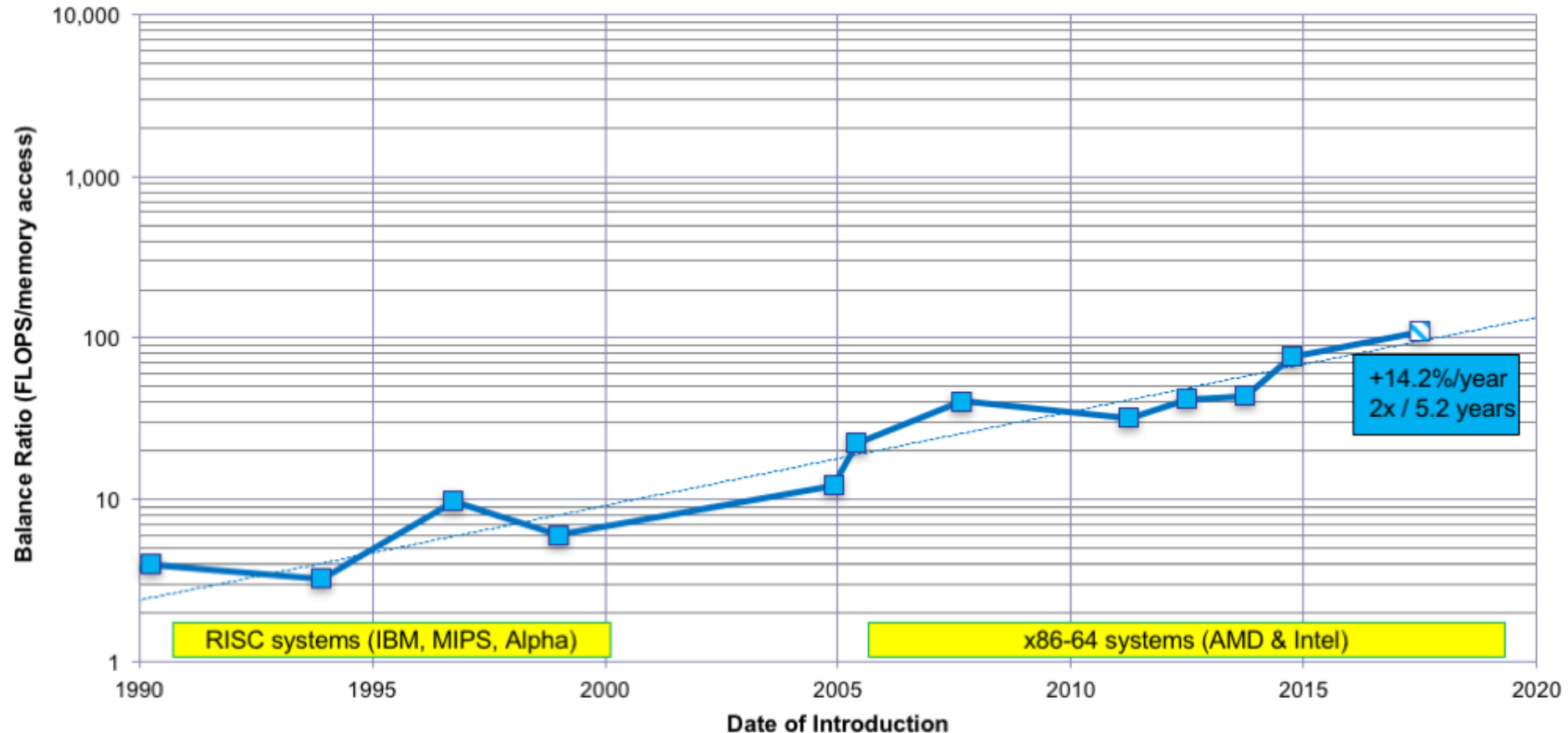
- Most programs have a high degree of locality
 - **spatial locality**: accessing things nearby previous accesses
 - **temporal locality**: reusing an item that was previously accessed
- Memory hierarchy use this to improve *average case*



Speed	1ns	10ns	100ns	10ms	10sec
Size	KB	MB	GB	TB	PB

Memory Bandwidth Gap

Memory Bandwidth is Falling Behind: (GFLOP/s) / (GWord/s)



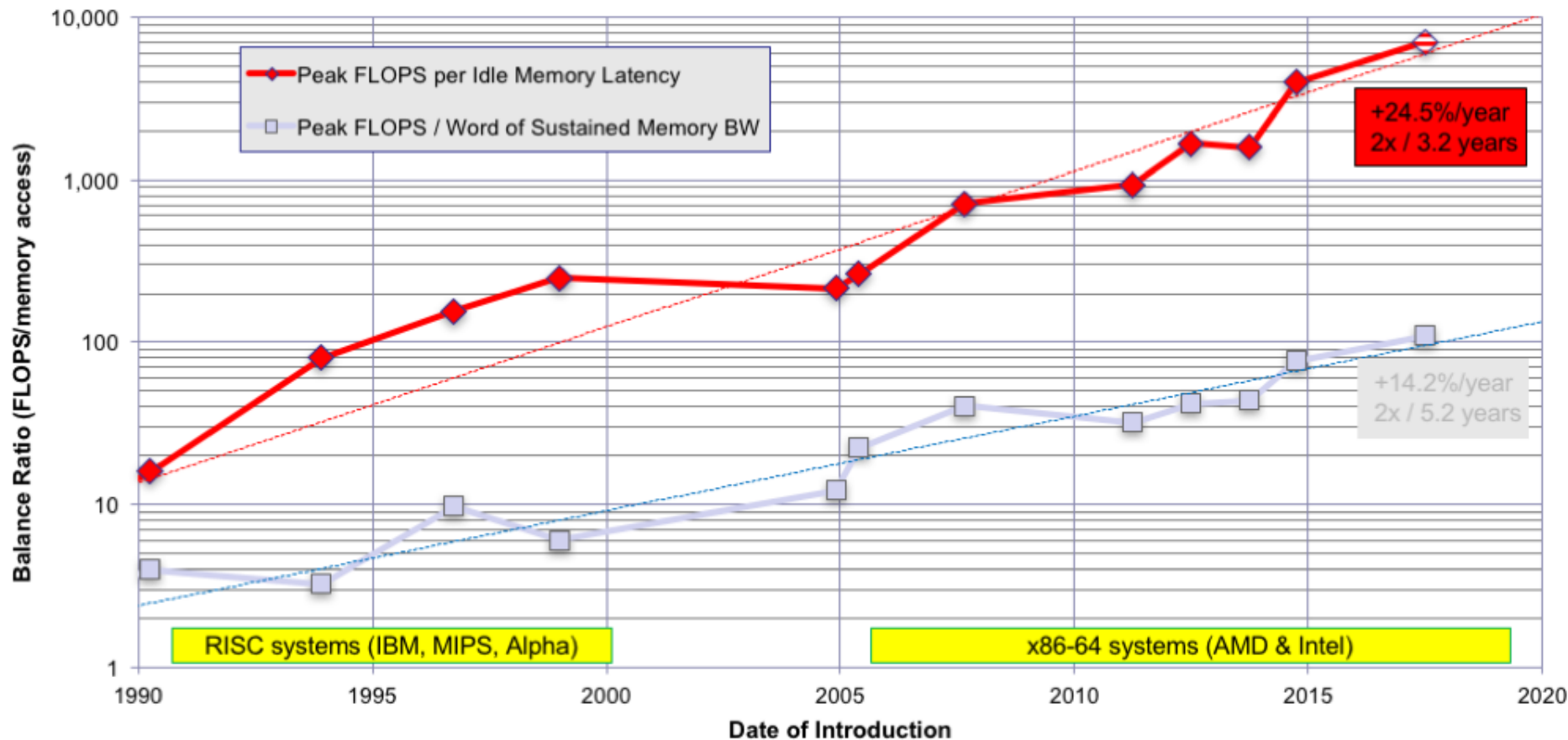
+14.2%/year
2x / 5.2 years

RISC systems (IBM, MIPS, Alpha)

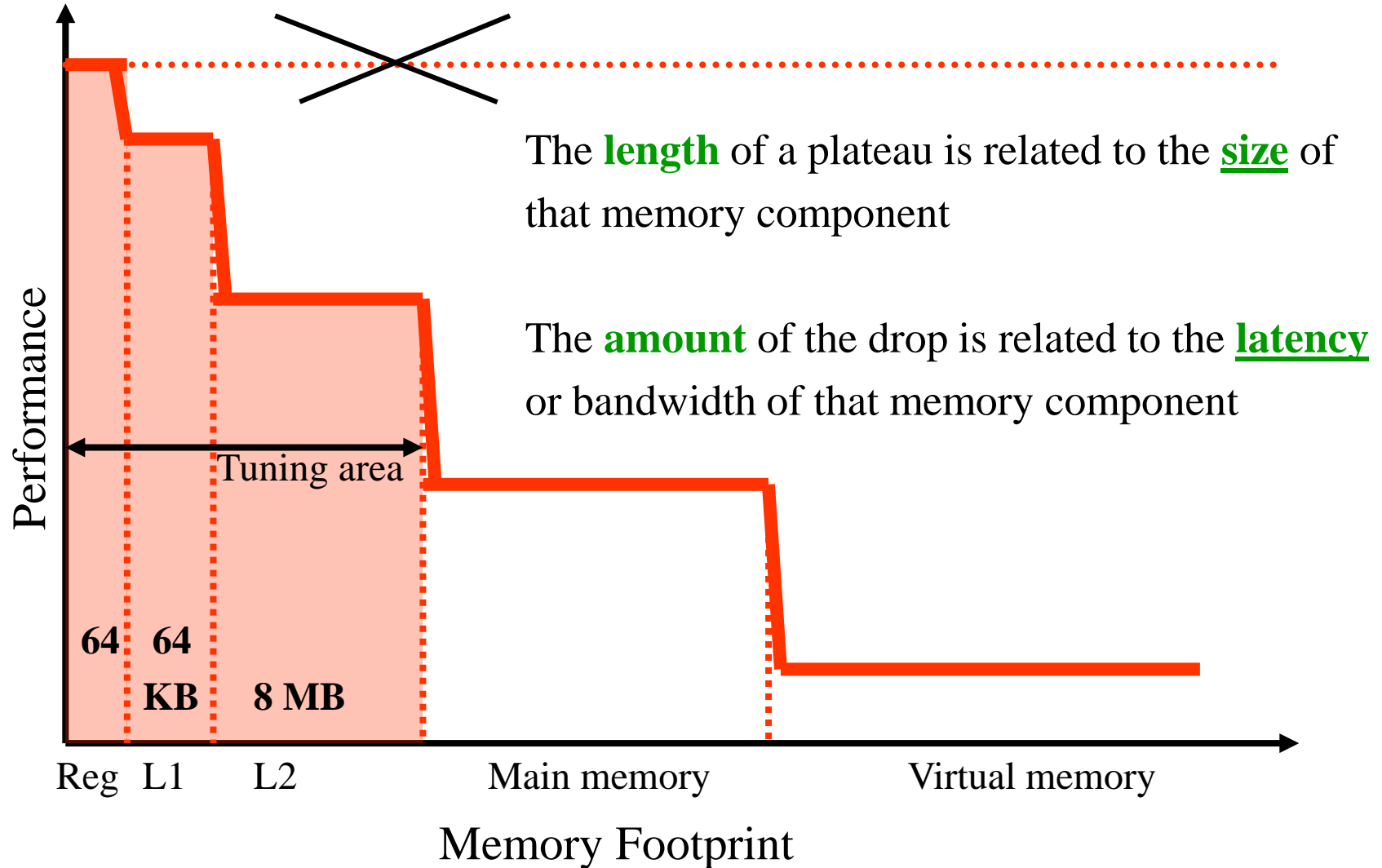
x86-64 systems (AMD & Intel)

Memory Latency Gap is Worse

Memory Latency is much worse: $(\text{GFLOP/s}) / (\text{Memory Latency})$

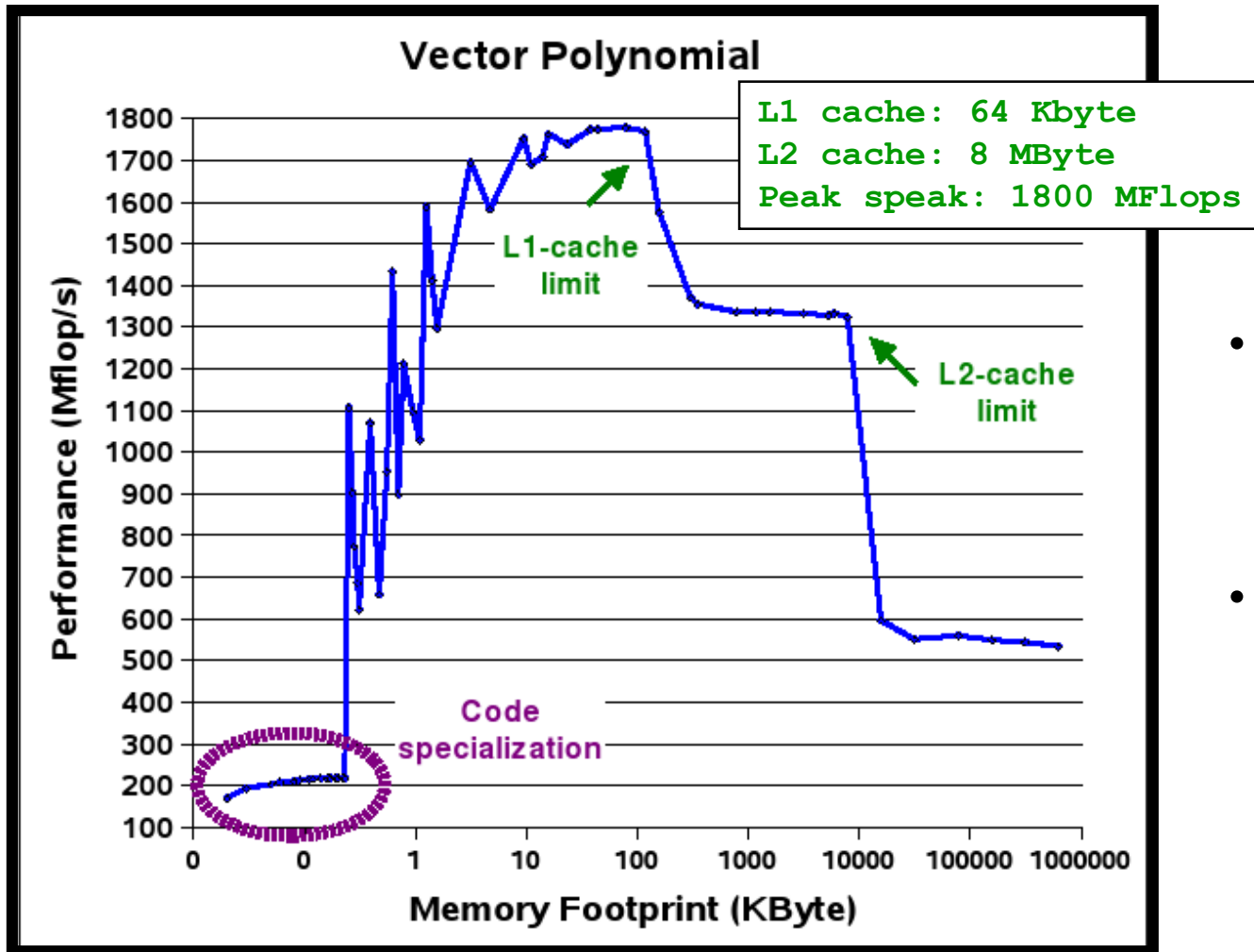


Performance is not Uniform



Example – 13th degree polynomial

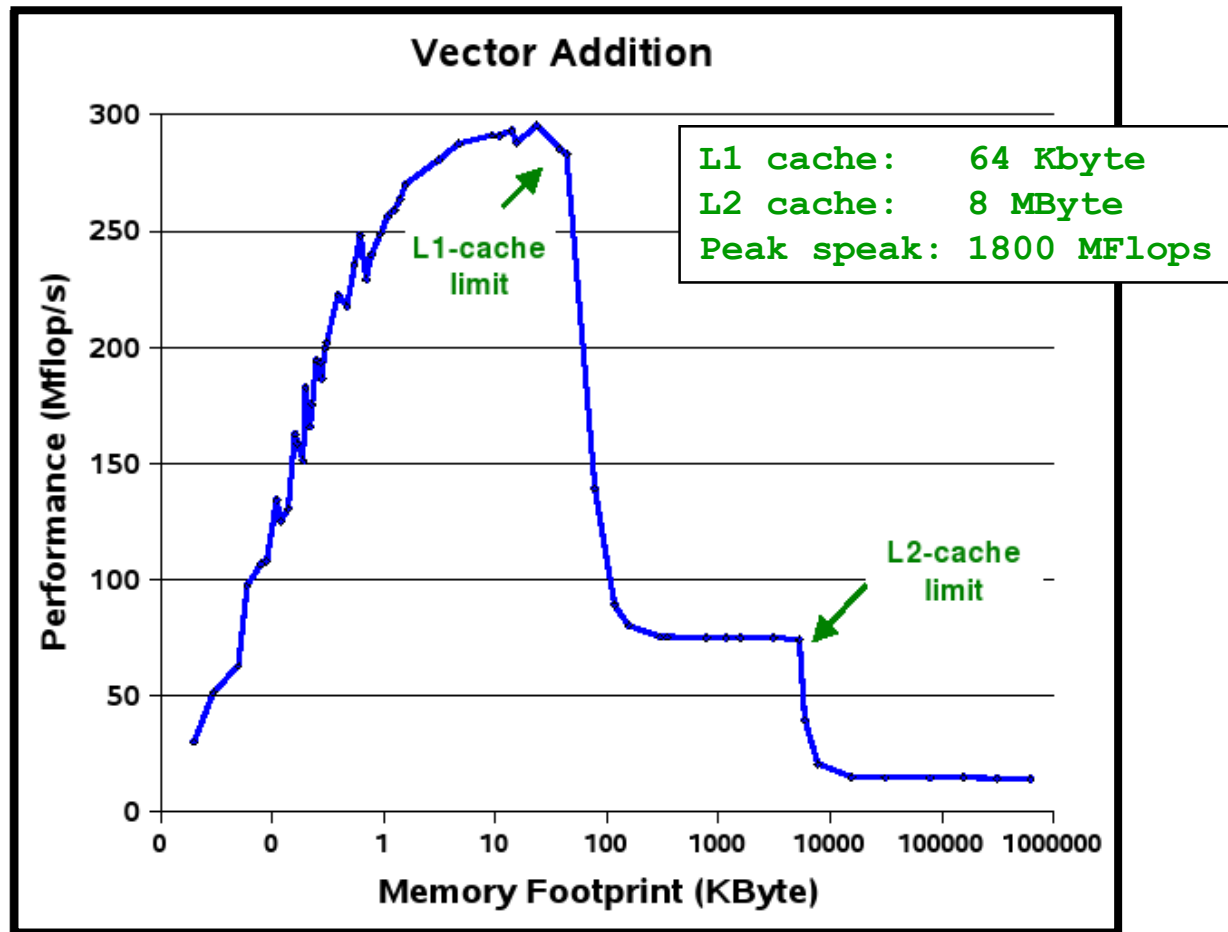
```
for ( i=0 ; i<vlen; i ++)  
    p[i]=c[0]+q[i]*(c[1]+q[i]*(c[2]+q[i]*(c[3]+ q[i] ...
```



- This operation is **CPU bound** i.e. there are much more floating point operations than memory references
- The system realizes over 98% of the absolute peak performance
- Note that's start-up effect and the performance drop for larger problems

Example – Vector Addition

```
for ( i=0 ; i<vlen; i ++)  
    p[i]= q[i] + r[i];
```



- This operation is memory bound i.e. there are much more memory references than floating point operations
- Note the start-up effect and the performance drop for larger problems

Outline

- Intro memory hierarchy

 Cache mappings

- Memory access
- Case study: Matrix-matrix multiplication

Cache Choices

- Modern systems use a wide variety of caches
- Typically there are at least 3 types of caches
 - Instruction cache
 - Data cache(s)
 - Page Address Cache (TLB)
- Some key design criteria
 - Size
 - Architecture/Mapping
 - Usage and cost are amongst the key decision factors

Cache Line Replacement

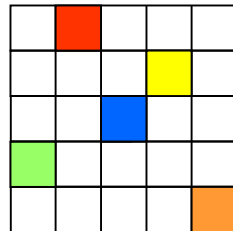
- Caches have a finite size
- A natural problem is how to replace data
- Three typical methods of doing this:
 - **Direct Mapped**
Location: Memory location modulo cache size
 - **Fully Associative**
Location: Least Recently Used (LRU, oldest cache line)
 - **X-way-Set Associative***
Location: Choose a set first; direct mapped in set

*Note: X is typically 2, 4 or 8

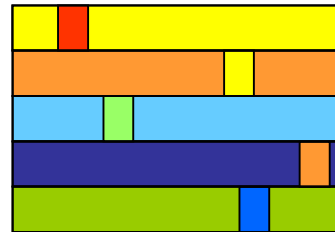
A Direct Mapped Cache

$$\text{Cache Location} = \text{Memory Address} \% \text{Cache Size}$$

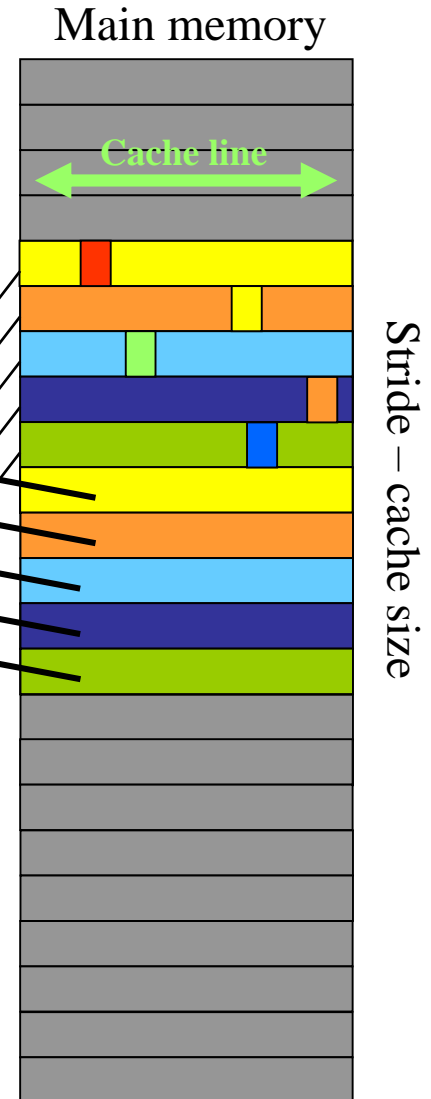
Virtual or physical address 



Register



Cache



- **Comments**

- Unit stride (=increment of 1) works well
- **Stride** is a multiple of cache size → **thrashing**
- It is therefore a function of the cache size
- Techniques like **padding** can reduce or eliminate trashing

Example Direct Cache Mapping

```
float a[4096], b[4096];  
for (i=0 ; i< 4096; i ++)  
    sum += a[i]*b[i];
```

Assumption:

- A cache line 8×4 Byte = 32 Byte

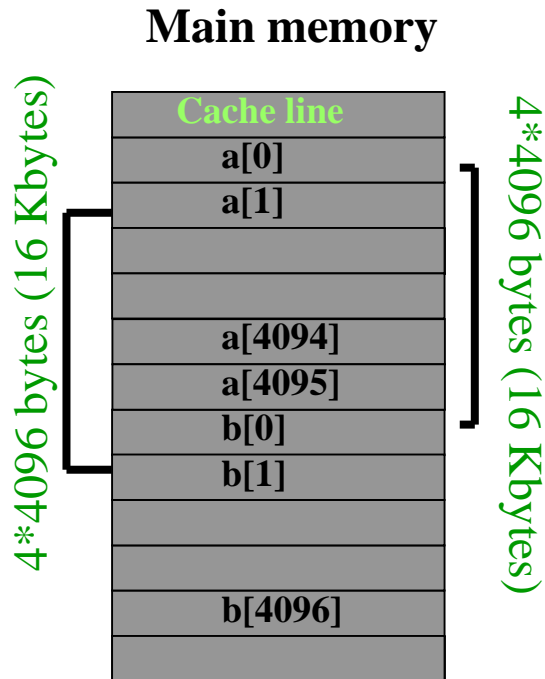
load a[0] into a reg.	miss : brings a[0] ... a[7] into cache
load b[0] into a reg.	miss : brings b[0] ... b[7] into cache
sum +=a[0]*b[0]	
load a[1] into a reg.	hit : brings a[1] still in cache
load b[1] into a reg.	hit : brings b[1] still in cache
sum +=a[1]*b[1]	
. . . etc . . .	
load a[7] into a reg.	hit : brings a[7] still in cache
load b[7] into a reg.	hit : brings b[7] still in cache
sum +=a[7]*b[7]	
load a[8] into a reg.	miss : brings a[8] ... a[15] into cache
load b[8] into a reg.	miss : brings b[8] ... b[15] into cache
sum +=a[8]*b[8]	
load a[9] into a reg.	hit : brings a[9] still in cache
load b[9] into a reg.	hit : brings b[9] still in cache
sum +=a[9]*b[9]	

Assume Different Memory Lay-out

```
float a[4096], b[4096];  
for (i=0 ; i< 4096; i ++)  
    sum += a[i]*b[i];
```

Assumption:

- A cache line 8*4 Byte = 32 Byte
- The cache size is 16 Kbyte



Cache

a[0]	a[7]
a[8]	a[15]
a[16] ...	
	a[4096]

16 Kbytes

The cache is completely filled in a[4096]

Every Reference is a Cache Miss!

```
float a[4096], b[4096];  
for (i=0 ; i< 4096; i ++)  
    sum += a[i]*b[i];
```

Assumption:

- A cache line 8×4 Byte = 32 Byte
- The cache size is 16 Kbyte

load a[0] into a reg.

miss: brings a[0] ... a[7] into cache

load b[0] into a reg.

miss: brings b[0] ... b[7] into cache

load of b[0] removes a[0]..a[7]

sum +=a[0]*b[0]

load a[1] into a reg.

miss: a[1] has just been removed

load of a[1] removes b[0]..b[7]

load b[1] into a reg.

miss: b[1] has just been removed

load of b[1] removes a[0]..a[7]

sum +=a[1]*b[1]

load a[2] into a reg.

miss: a[2] has just been removed

load of a[2] removes b[0]..b[7]

load b[2] into a reg.

miss: b[2] has just been removed

load of b[2] removes a[0]..a[7]

sum +=a[2]*b[2]

Padding – Change Address

```
float a[4096], padd, b[4096];  
for (i=0 ; i< 4096; i ++)  
    sum += a[i]*b[i];
```

Main memory

Cache line
a[0]
a[1]
a[4094]
a[4095]
b[0]
b[1]
b[4096]

No more conflicts !

Cache

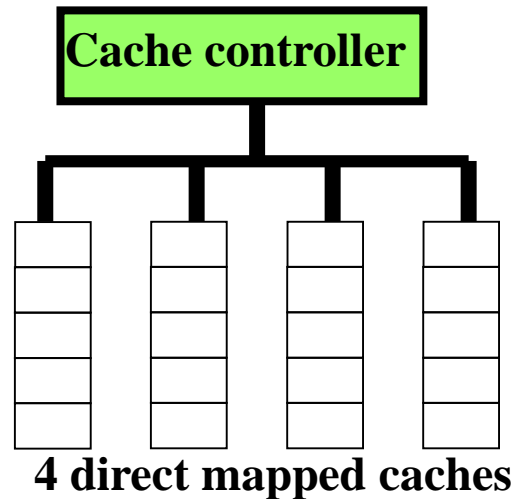
a[0]	a[7]
b[0] ...	b[7]

16 Kbytes

The cache can use “a” and “b”

X-way Set Associative

- Alternative to the direct mapped cache → have a bunch of them
- Example: **4-way set associative**



- Comments

- Choice of a set can be random or last recently used.
- The more sets, the better
- Issues
 - Size per set is $1/x$ of total
 - Higher chance of thrashing on one set
 - More loads/stores than sets → behavior similar to direct mapped, but set size is smaller

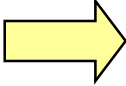
Choose a set and within that set:

Cache Location = Memory Address % Cache Size

Virtual or physical address

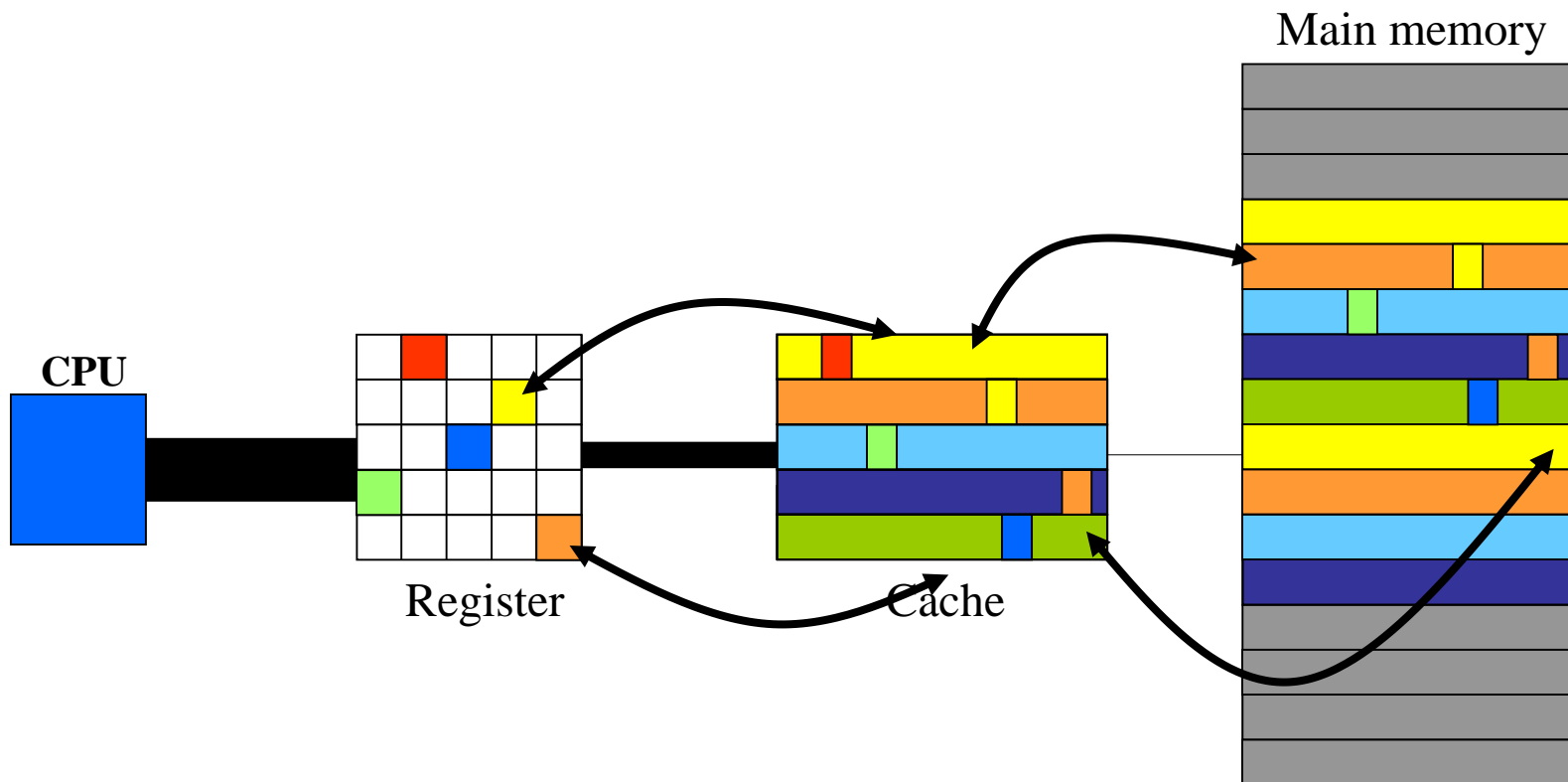


Outline

- Intro memory hierarchy
- Cache mappings
-  Memory access
- Case study: Matrix-matrix multiplication

Cache lines

- For **good performance**, it is crucial to use the cache(s) in the intended (=optimal) way
- Recall that the unit of transfer is a cache line
- A cache line is a linear structure i.e. it has a fixed length (in bytes) and a starting address in memory



Cache Line Utilization

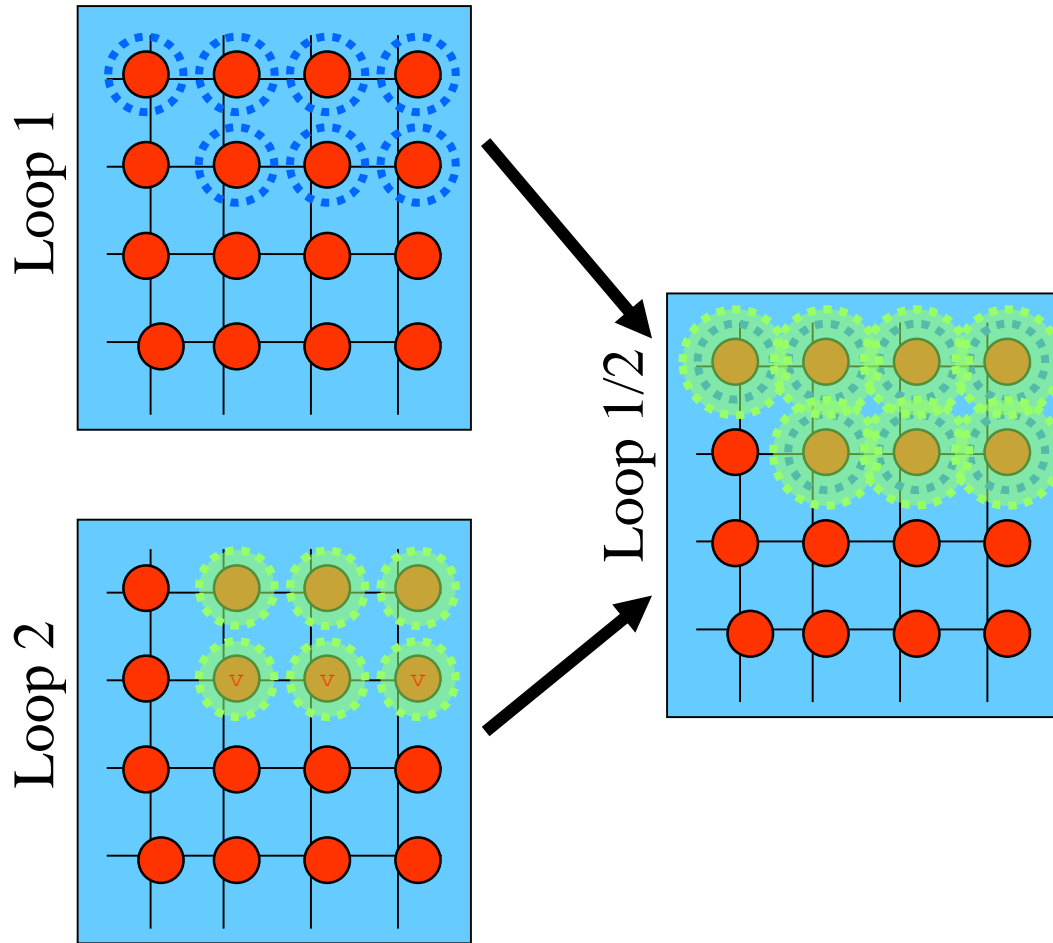
- Two Key **Rules for Performance** – **Maximize**
- **Spatial Locality** – Use all data in one cache line

This strongly depends on the storage of your data and the access patterns

- **Temporal Locality** – Re-use data in a cache line

This mainly depends on the algorithm used

Cache Line Re-Use



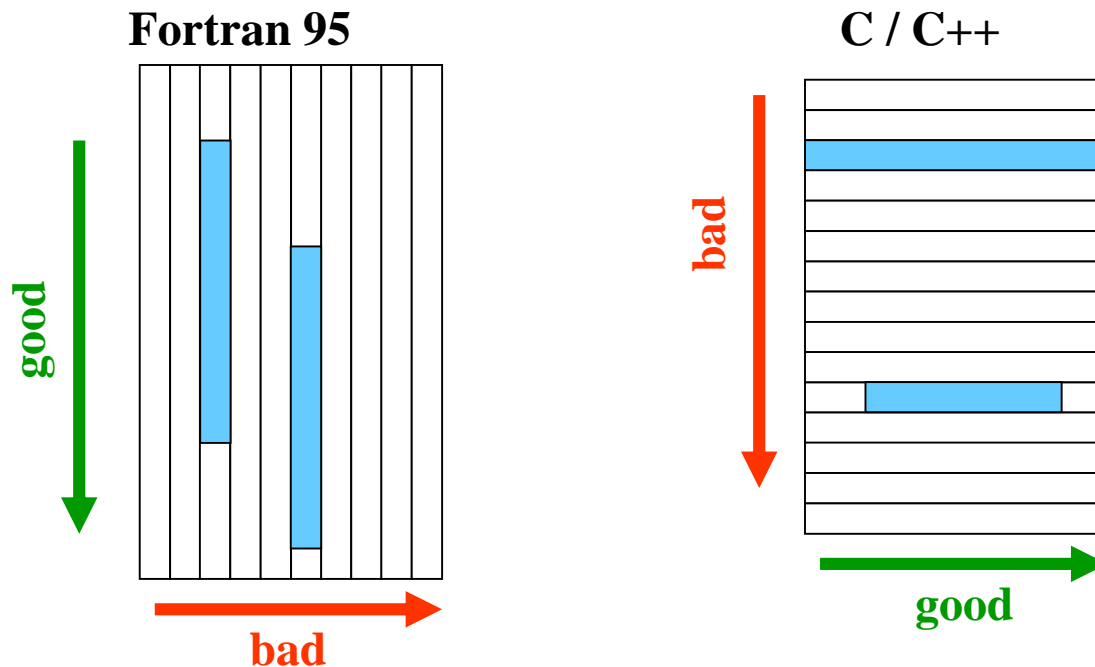
- On the left we show a typical vector style of coding
- It is not a good approach for cache based systems: all grid elements have to be reloaded for each loop
- It is more beneficial to pre-calculate expression on the already loaded grid points

Cache Types

- Generally, one can distinguish 3 different caches:
 - **Data cache**
Most important in scientific applications
 - **Instruction Cache**
More important in business applications
 - **TLB Cache** (to cache address translation)
Important for every application
- If these caches are not used in the **expected way**, bad things can happen
- So what is the expected way?

Memory Access

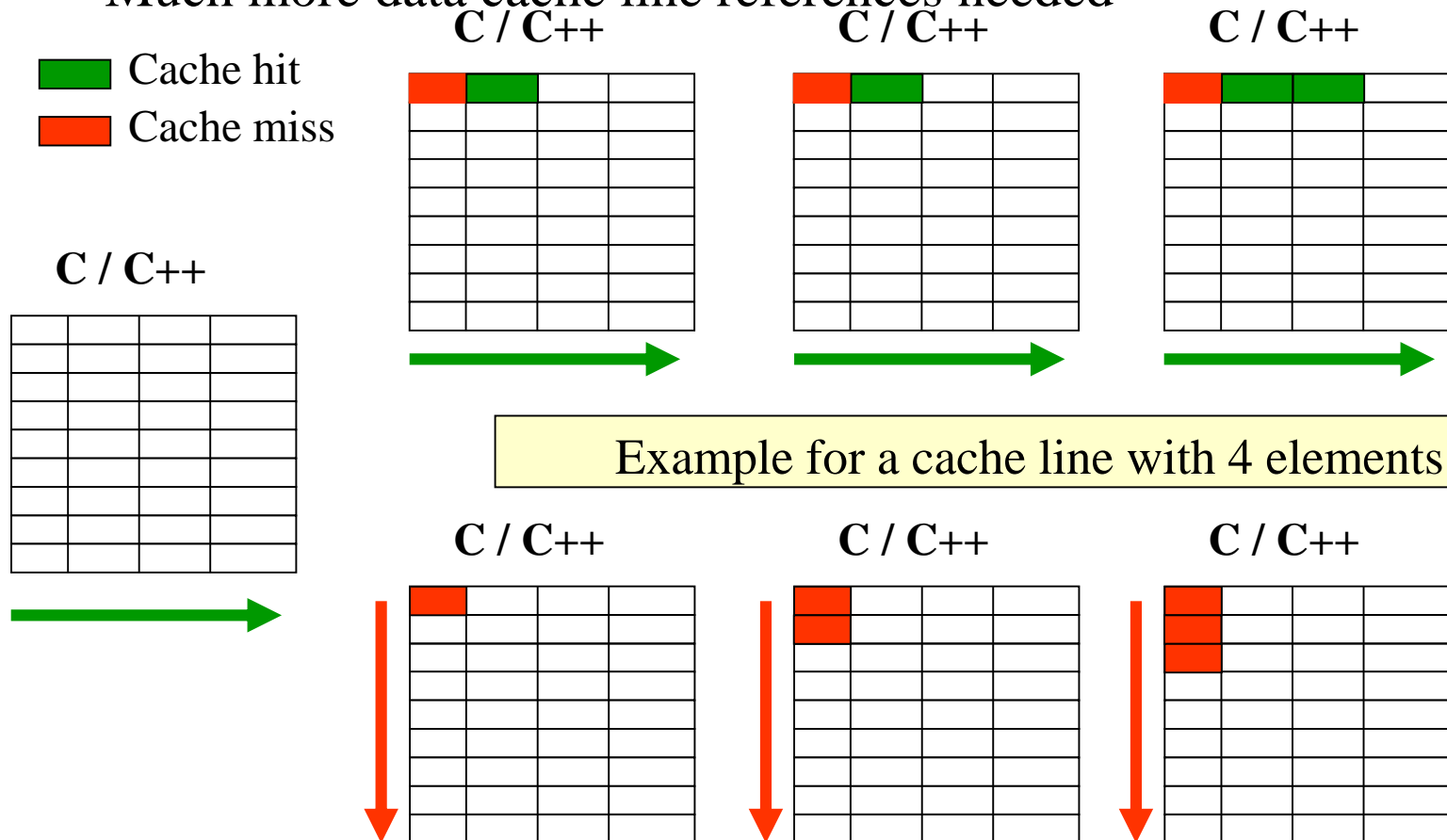
- Memory has always a 1D linear structure
- Access to multi-dimensional arrays depends on the way data is stored
- This is language dependent:



Bad Memory Access has a Huge Impact on Performance

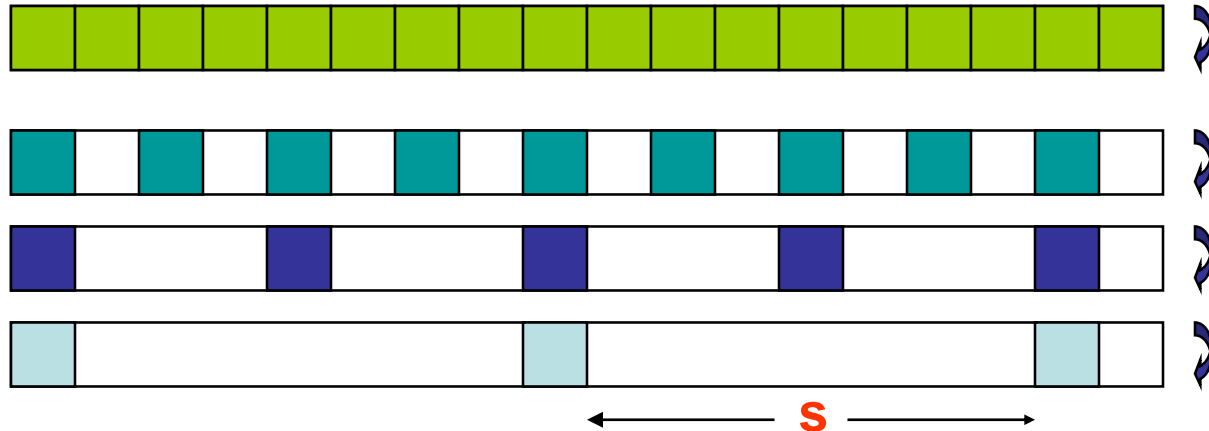
Non-unit Stride is always Bad

- If one follows the storage order in memory, all is well; This is called unit stride or (stride=1)
- If data is accessed with **non-unit stride**:
 - Much more data cache line references needed



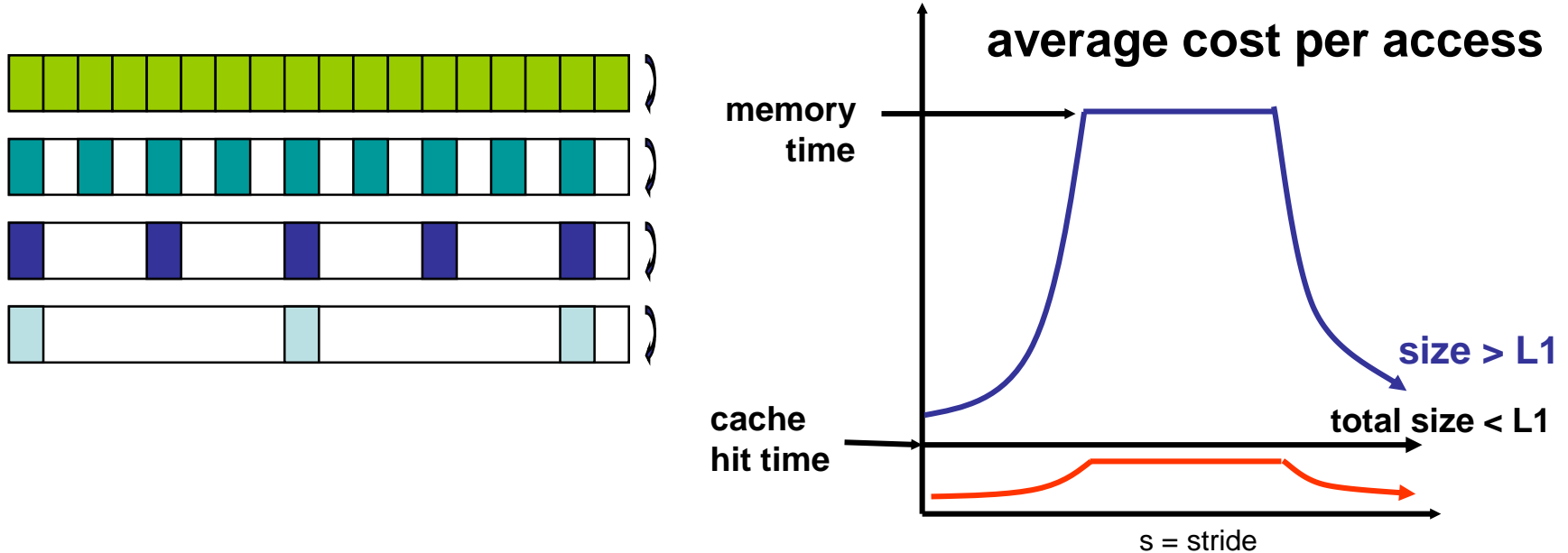
Experimental Study of Memory

- Micro benchmark 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 and store A[i] from memory (4 Bytes)
```

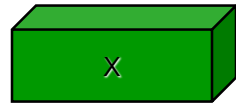
Experimental Study of Memory: What to expect



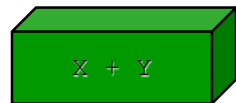
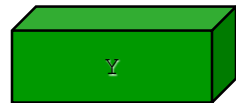
- Consider the average cost per load
 - **Plot one line** for each array length, **time vs. stride**
 - **Small stride is best**: if cache line holds 4 words, at most $\frac{1}{4}$ miss
 - If array is smaller than a given cache, all those accesses will hit (after the first run, which is negligible for large enough runs)
 - Picture assumes only one level of cache
 - Values have gotten more difficult to measure on modern procs

SIMD: Single Instruction, Multiple Data

- Scalar processing
 - **traditional mode**
 - one operation **produces** one result



+



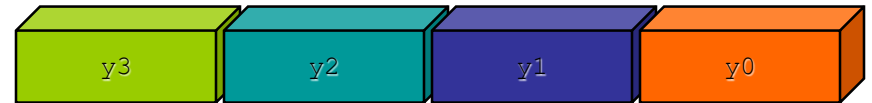
- SIMD processing
 - **with SSE / SSE2**
 - **SSE = streaming SIMD extensions**
 - one operation **produces** multiple results

X



+

Y

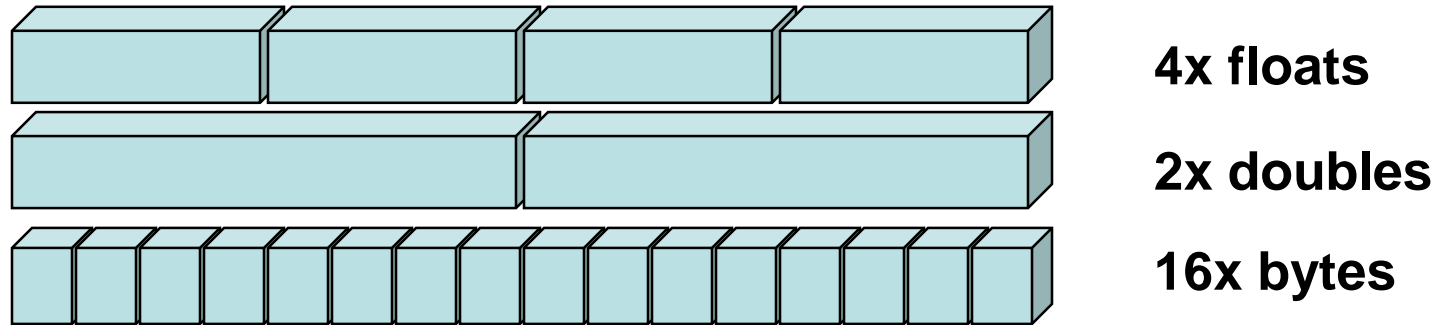


X + Y



SSE / SSE2 SIMD on Intel

- SSE2 data types: anything that fits into 16 bytes, e.g.,



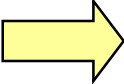
- Instructions perform add, multiply etc. on all the data in this 16-byte register in parallel
- Challenges: Need to be contiguous in memory and aligned
- Similar on GPUs, vector processors (but many more simultaneous operations)
- Oct 2012: **Intel Xeon Phi** with single-instruction-multiple-data (**SIMD**) instructions using **64 bytes** (512-bits) = 8 doubles

What does this mean to you?

- In addition to SIMD extensions, the processor may have other special instructions
 - Fused Multiply-Add (FMA) instructions:
$$x = y + c * z$$
is so common some processor execute the multiply/add as a single instruction, at the same rate (bandwidth) as + or * alone
- In theory, the compiler understands all of this
 - When compiling, it will rearrange instructions to get a good “schedule” that maximizes pipelining, uses FMAs and SIMD
- But in practice the compiler may need your help
 - Choose a different compiler, optimization flags, etc.
 - Rearrange your code to make things more obvious
 - Using special functions (“intrinsics”) or write in assembly ☹

Outline

- Intro memory hierarchy
- Cache mappings
- Memory access

 Case study: Matrix-matrix multiplication

Lessons

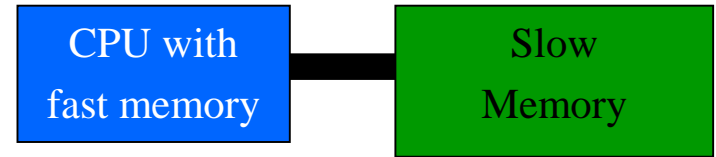
- **Actual performance** of a simple program can be a complicated function of the architecture
 - Slight changes in the architecture or program change the performance **significantly**
 - To write **fast programs**, need to consider architecture
 - True on sequential or parallel processor
 - We would like simple models to help us design efficient algorithms
- We will illustrate with a common technique for improving cache performance, called **blocking**
 - Idea: used **divide-and-conquer** to define a problem that fits in register/L1-cache/L2-cache

Case study: Matrix-Matrix Multiplication

- Case study: **Matrix-Matrix Multiplication**
 - Use of performance models to understand performance
 - Simple cache model
- Why Matrix Multiplication?
 - Appears in many linear algebra algorithms
- Optimization ideas can be used in other problems
- The best case for optimization payoffs
- The most-studied algorithm in high performance computing

Case study: Matrix Multiplication

- Assume just **2 levels** in the **memory** hierarchy, **fast** and **slow**



- All data initially in slow memory
 - m = number of memory elements (words) moved between fast and slow memory

- t_m = time per **slow memory** operation

- f = number of arithmetic operations

- t_f = time per **arithmetic operation** in **fast memory** $\ll t_m$

Computational Intensity: Key to algorithm efficiency

- $q = f / m$ average number of flops per slow memory access

- Minimum possible time = $f * t_f$ when all data in fast memory

- Actual time

- $f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * m/f) = f * t_f * (1 + \boxed{t_m/t_f} * 1/q)$

Machine Balance: Key to machine efficiency

- Larger q** means time closer to minimum $f * t_f$

- $q \geq t_m/t_f$ needed to get at least half of peak speed

Warm up: Vector-vector operation

- **Assumption:** Fast memory large enough to store 2 vectors, “a” is a scalar, x, y are vectors

```
read x(1:n) into fast memory
read y(1:n) into fast memory
for j = 1:n
    y(i) = y(j) + a*x(i)
write y(1:n) back into slow memory
```

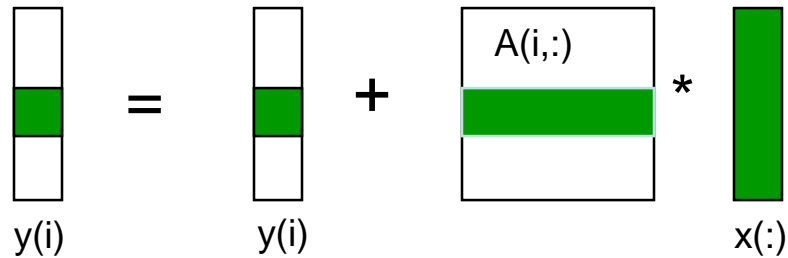
$$\begin{matrix} \square \\ \text{green} \\ \square \end{matrix} = \begin{matrix} \square \\ \text{green} \\ \square \end{matrix} + a * \begin{matrix} \square \\ \text{green} \\ \square \end{matrix}$$

$y(i)$ $y(i)$ $x(i)$

- m = number of slow memory accesses $\rightarrow m = 3n$
- f = number of floating point operations $\rightarrow 2n$
- q = average number of flops per slow memory access $= f / m = 2/3$
- Vector-vector addition is dominated by slow memory access

Warm up: Matrix-vector multiplication

```
// implements  $y = y + A*x$   
for i = 1:n  
    for j = 1:n  
         $y(i) = y(i) + A(i,j)*x(j)$ 
```



Warm up: Matrix-vector multiplication

Assumption: Fast memory large enough to store 2 vectors y and x and one row of matrix A

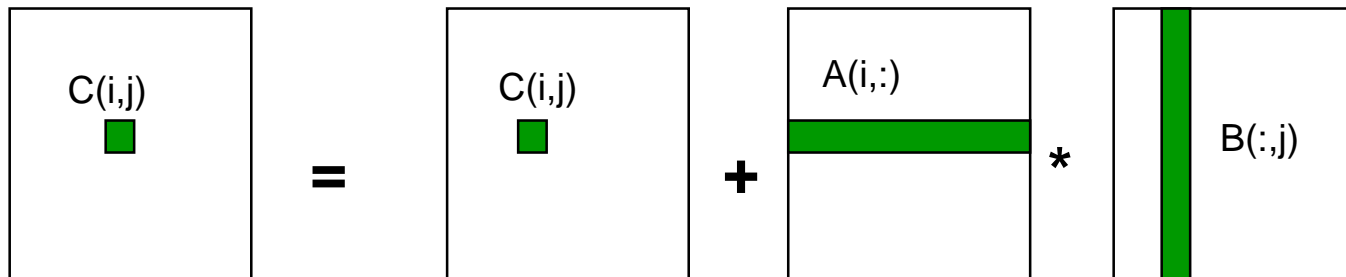
```
read x(1:n) into fast memory
read y(1:n) into fast memory
for i = 1:n
    read row i of A into fast memory
    for j = 1:n
        y(i) = y(i) + A(i,j)*x(j)
write y(1:n) back to slow memory
```

- m = number of slow memory refs $\rightarrow m = 3n + n^2$
- f = number of arithmetic operations $\rightarrow f = 2n^2$
- $q = f / m \approx 2$
- Matrix-vector multiplication is limited by slow memory speed

Naive Matrix-Matrix Multiplication

```
// computes C = C + A*B
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      c(i,j) = c(i,j) + a(i,k) * b(k,j)
```

- Algorithm has $2*n^3 = O(n^3)$ Flops and operates on $4*n^2$ words of memory \rightarrow q potentially as **large** as $2*n^3 / 4*n^2 = \mathbf{O(n)}$



Naive Matrix-Matrix Multiplication

Assumption: Fast memory large enough to store one row of the matrices A, B and C.

```
// implements C = C + A*B
for i = 1 to n
  read row i of A into fast memory
  for j = 1 to n
    read scalar C(i,j) into fast memory
    read column j of B into fast memory
    for k = 1 to n
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
    write scalar C(i,j) back to slow memory
```

- Number of **slow memory** references on unblocked matrix multiply

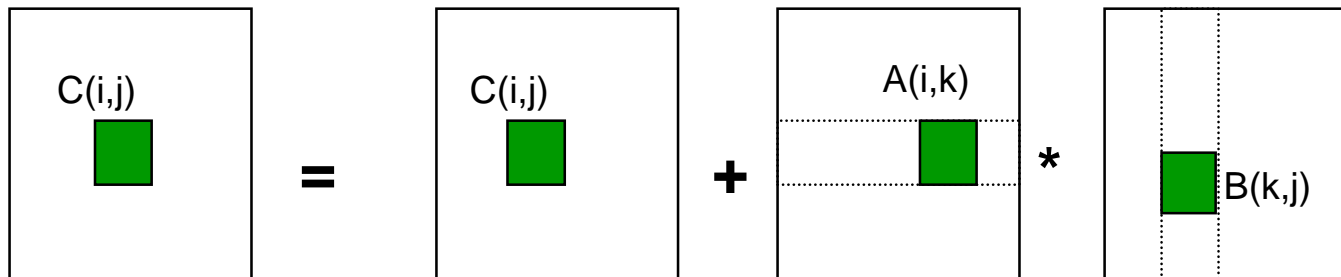
$$\begin{aligned} m &= n^3 && \text{to read each column of B } n^2 \text{ times} \\ &+ n^2 && \text{to read each row of A once} \\ &+ 2n^2 && \text{to read and write each element of C once} \\ &= \underline{n^3 + 3n^2} \end{aligned}$$

So $q = f / m = 2n^3 / (n^3 + 3n^2) \approx 2$ for large n , **no improvement** over matrix-vector multiply, but on previous slide it was $O(n)$!

Blocked Matrix Multiply

- **Assumption:** Fast memory large enough to store small subblocks row of the matrices A, B and C.

```
// Consider A,B,C to be n-by-n matrix viewed as
// N-by-N matrices of b-by-b subblocks where
// b=n/N is called the block size
for I = 1 to N
  for J = 1 to N
    read block C(I,J) into fast memory
    for K = 1 to N
      read block A(I,K) into fast memory
      read block B(K,J) into fast memory
      // do a matrix multiply on blocks
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
    write block C(I,J) back to slow memory
```



Blocked Matrix Multiply

- Recall:
 - m is amount **memory traffic** between slow and fast memory
 - matrix has nxn elements, and **NxN blocks** each of size **bxb**
 - f is **number of floating point** operations, **2n³** for this problem
 - $q = f / m$ is the measure of algorithm efficiency in the memory model
- So:
$$\begin{aligned} \mathbf{m} &= \mathbf{N * n^2} \quad \text{read each block of B } N^3 \text{ times } (N^3 * b^2 = N^3 * (n/N)^2 = N * n^2) \\ &\quad + \mathbf{N * n^2} \quad \text{read each block of A } N^3 \text{ times} \\ &\quad + \mathbf{2n^2} \quad \text{read and write each block of C once} \\ &= \mathbf{(2N + 2) * n^2} \end{aligned}$$
- **The computational intensity is**
$$\mathbf{q = f / m = 2n^3 / ((2N + 2) * n^2) \sim n / N = b \text{ for large } n}$$

So we can improve performance by **increasing** the blocksize (**cachesize**) b and can be much faster than matrix-vector multiply (q=2)

Using Analysis to Understand Machines

The blocked algorithm has computational intensity $q \approx b$

- The larger the block size, the more efficient our algorithm will be
- Limit: All three blocks from A,B,C must fit in fast memory (cache), so we cannot make these blocks arbitrarily large
- Assume your fast memory has size M_{fast}

$$3b^2 \leq M_{\text{fast}}, \quad \text{so } q \approx b \leq \sqrt{M_{\text{fast}}/3}$$

- To build a machine to run matrix multiply at 1/2 peak arithmetic speed of the machine, we need a fast memory of size

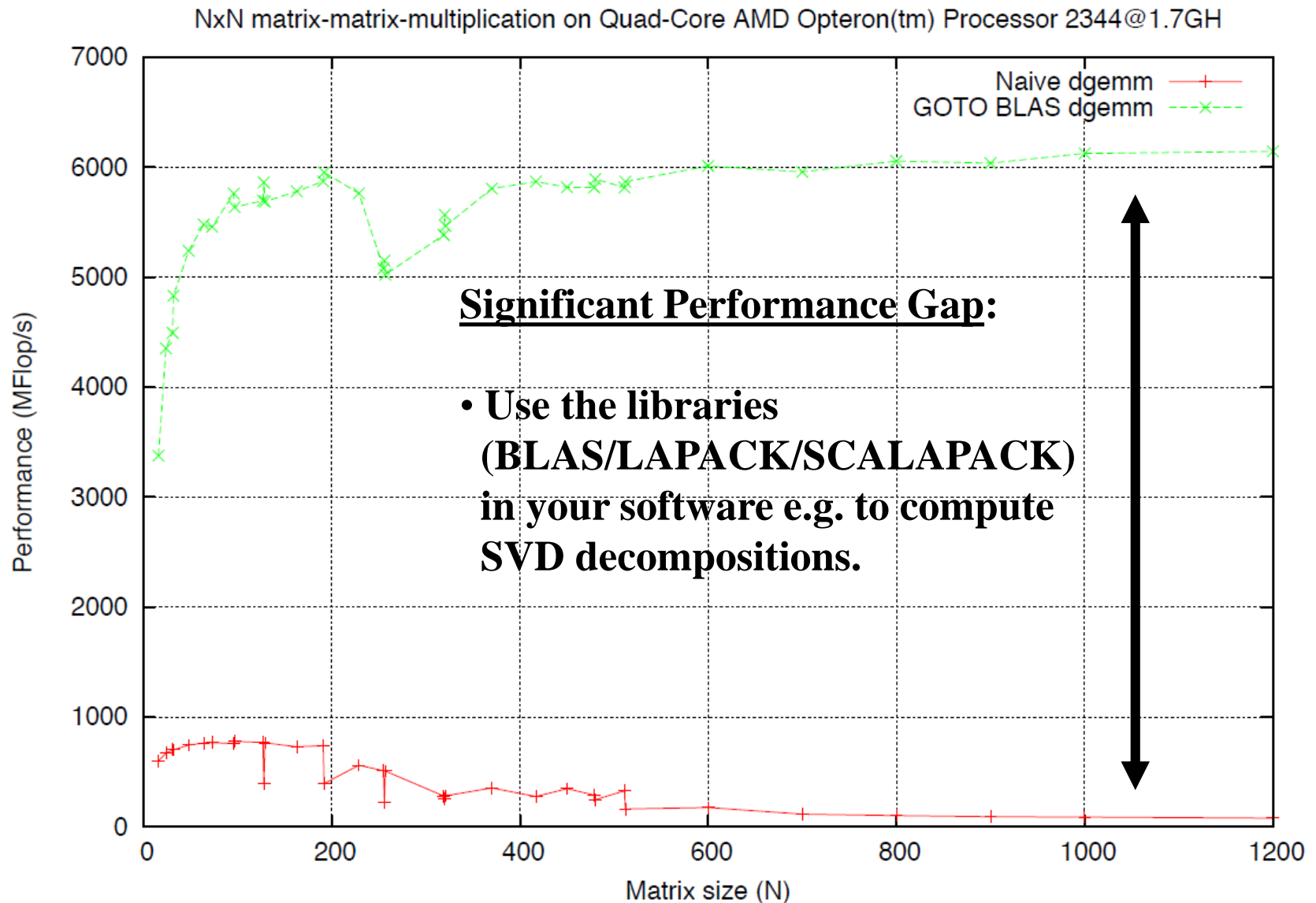
$$M_{\text{fast}} \geq 3b^2 \approx 3q^2 = 3(t_m/t_f)^2$$

- What if more levels of memory hierarchy?
 - Apply blocking recursively, once per level

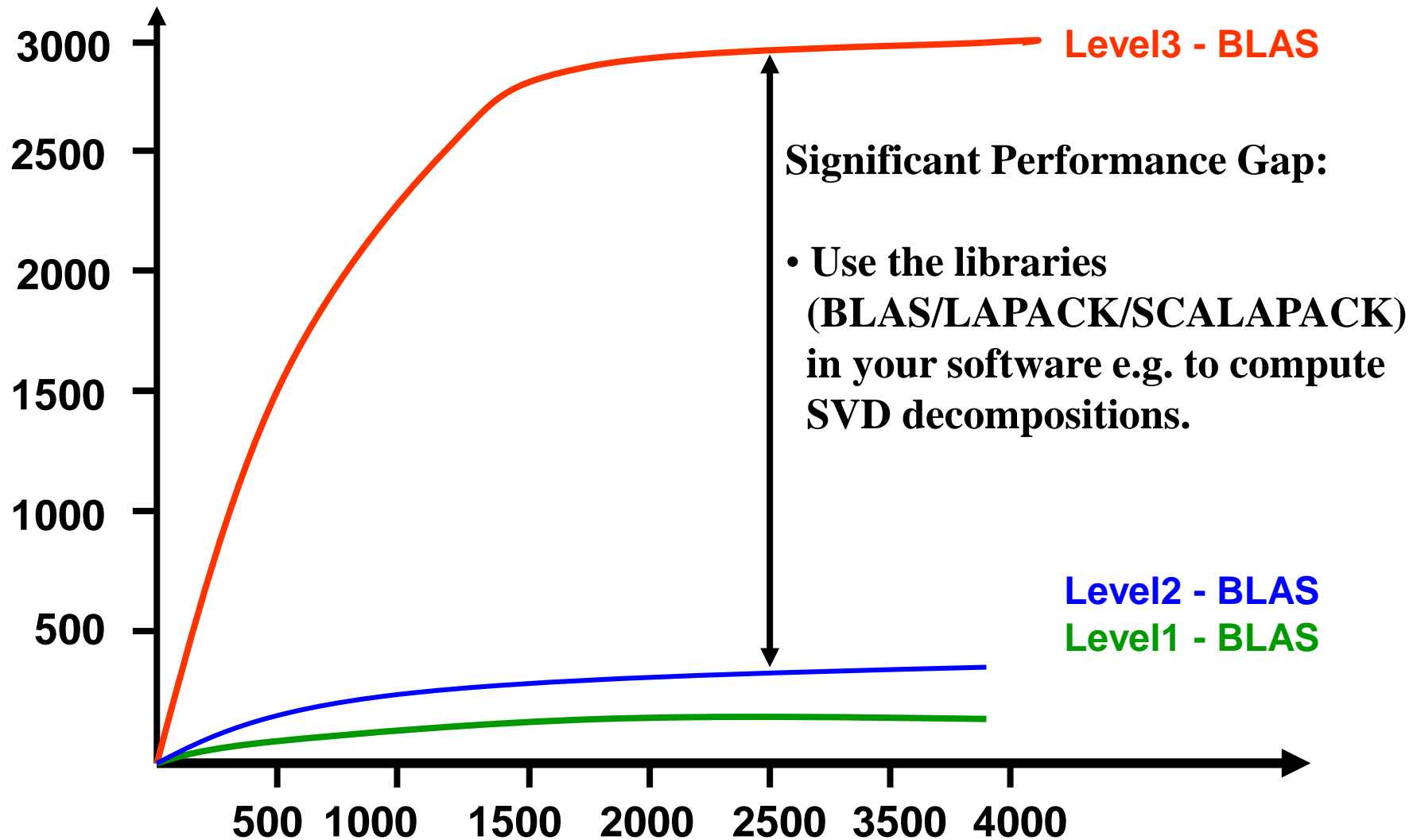
Use optimized libraries for matrix-/vector operations

- Industry library standard interface: www.netlib.org/blas
- Vendors, others supply highly optimized implementations
 - **BLAS – Basic Linear Algebra Subroutines**
 - **BLAS1 (1970s):**
 - vector operations: dot product, saxpy ($y=a*x+y$), etc
 - $m=2*n$, $f=2*n$, $q \sim 1$ or less
 - **BLAS2 (mid 1980s)**
 - matrix-vector operations: matrix vector multiply, etc
 - $m=n^2$, $f=2*n^2$, $q \sim 2$, less overhead
 - somewhat faster than BLAS1
 - **BLAS3 (late 1980s/earlier 90)**
 - matrix-matrix operations: matrix-matrix multiply, etc
 - $m \leq 3n^2$, $f=O(n^3)$, so $q=f/m$ can possibly be as large as n , so BLAS3 is potentially much faster than BLAS2
- Good algorithms used BLAS3 when possible (LAPACK & ScaLAPACK)
- **LAPACK: Linear Algebra Package**
ScaLAPACK: Scalable Linear Algebra Packages (parallel)

Real Matrix-Matrix Performance on CUB Cluster



BLAS Performance on Xeon SSE2 with 1.5 GHz



A last note on the complexity of matrix multiply

- The traditional algorithm (with or without blocking) has $O(n^3)$ flops
- $n \times n$ Matrix = 2×2 Matrix with $n/2 \times n/2$ block matrices

```
Let M = | m11 m12 | = | a11 a12 | * | b11 b12 |
         | m21 m22 |   | a21 a22 |   | b21 b22 |
         m11 = a11*b11 + a12*b21,  m12 = a11*b12 + a12*b22
         m21 = a21*b11 + a22*b21,  m22 = a21*b12 + a22*b22
```

- Reduction to four sub problems with
 - 2 multiplications von $n/2 \times n/2$ matrices
 - 1 addition von $n/2 \times n/2$ matrices
 - Total: 8 Mult, 4 Adds
- **Complexity:** $T(n) = 8 * T(n/2) + \theta(n^2)$
 $\quad \quad \quad \text{\#subproblems / size of subproblems / complexity for addition}^6$
 $= 8 (n/2)^3 = n^{\log_2 8} = n^3$
 $= \theta(n^3) \quad \quad \text{No reduction!}$

A last note on the complexity of matrix multiply

- Upper complexity bound for matrix-matrix multiply?
- Lower complexity bound for matrix-matrix multiply?

Complexity of Strassen's Matrix Multiply

- Strassen discovered an algorithm with asymptotically lower flops
 - $O(n^{2.81})$
- Consider a 2x2 matrix multiply, normally takes 8 multiplies, 4 adds
 - Strassen does it with **7 multiplies and 18 adds**

```
Let M = | m11 m12 | = | a11 a12 | * | b11 b12 |
        | m21 m22 |   | a21 a22 |   | b21 b22 |
```

```
Let p1 = (a12 - a22) * (b21 + b22)    p5 = a11 * (b12 - b22)
    p2 = (a11 + a22) * (b11 + b22)    p6 = a22 * (b21 - b11)
    p3 = (a11 - a21) * (b11 + b12)    p7 = (a21 + a22) * b11
    p4 = (a11 + a12) * b22
```

```
Then m11 = p1 + p2 - p4 + p6
    m12 = p4 + p5
    m21 = p6 + p7
    m22 = p2 - p3 + p5 - p7
```

Complexity:

$$T(n) = 7 * T(n/2) + \theta(n^2)$$

#subproblems / size of subproblems / complexity for addition

$$= 7 (n/2)^3 = n^{\log_2 7} + \theta(n^2)$$

$$= \theta(n^{2.81})$$

Summary

- Today's memory hierarchy is very complex
- Performance often depends on the problem size
- Caches play a crucial role in performance
- Key to using memory in the right way:
 - Access your data in storage order
 - Re-use data where possible
- Spatial and temporal locality
- Use optimized libraries as often as possible in your code.