
Hierarquia de memória, modelagem de desempenho Otimização da Multiplicação de Matrizes

Slides tomados de James Demmel

http://www.cs.berkeley.edu/~demmel/cs267_Spr16/

, Computer Systems: a programmer's perspective

A aula de hoje

- A maioria das aplicações executam a $< 10\%$ do desempenho “pico” do sistema
- Intensidade computacional $q=f/m$: medida da eficiência do algoritmo
- Custo médio de operações aritméticas (f) por número de operações na memória lenta (m)
- Queremos $q \geq t_m/t_f$ (*machine balance – medida da eficiência da máquina*) para que o desempenho do programa dependa unicamente de t_f e f

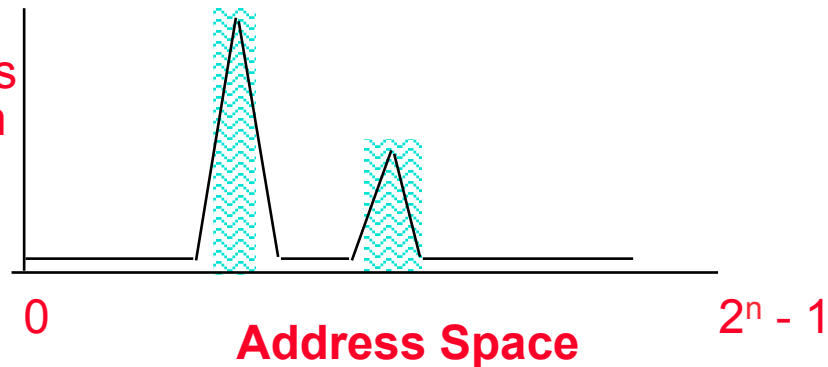
Para isto procuramos diminuir os acessos a memória lenta.

Algoritmo ingenuo de Mult. Matrices: $q = 2$

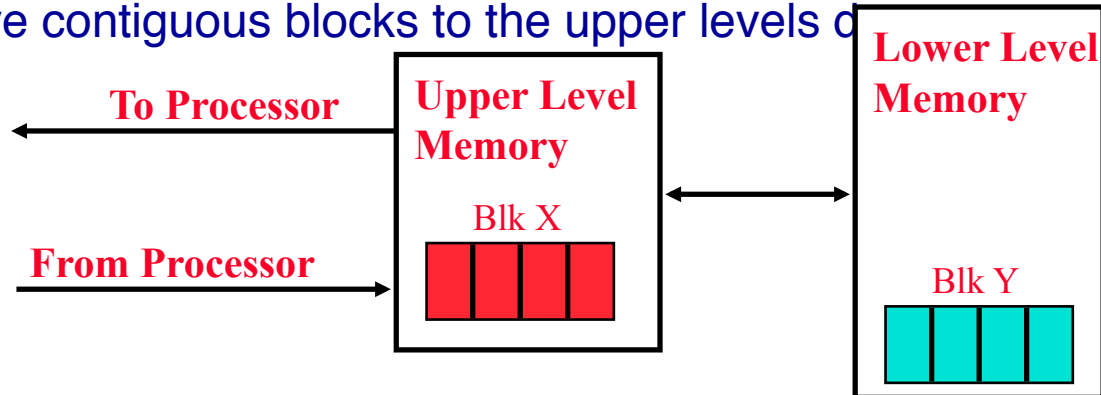
- Tijolamento (blocking/tiling) é uma abordagem básica para aumentar q (**aumentar localidade!**)
 - As técnicas são gerais mas os detalhes dependem da arquitetura (o tamanho do bloco)
 - Técnicas similares possíveis em outras estruturas de dados e algoritmos

Why Does Caching Help? Locality!

Probabilidade de referencia: fração das referencias que caem em um endereço x



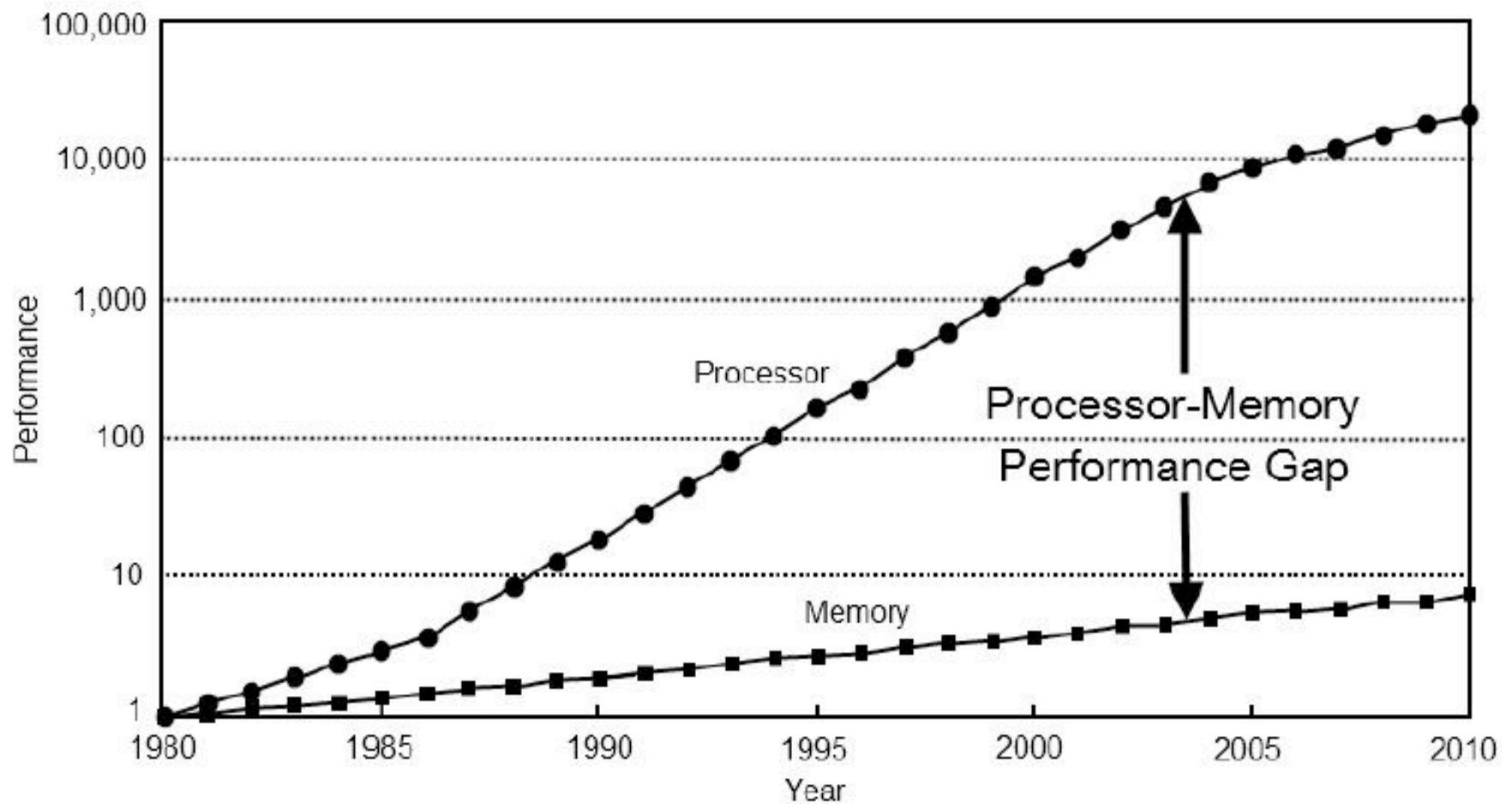
- **Temporal Locality** (Locality in Time): Se acessamos uma localização no passado provavelmente vamos acessá-la novamente no futuro
 - Keep recently accessed data items closer to processor
- **Spatial Locality** (Locality in Space): Provavelmente as próximas localizações acessadas estarão perto do endereço acessado (ex. Array)
 - Move contiguous blocks to the upper levels of memory



Motivação

- A maioria das aplicações executam a $< 10\%$ do desempenho pico de um sistema
 - Pico: o máximo que o hardware consegue fisicamente executar
- O desempenho do código que executa 1 processador com frequência é somente de 10-20% do pico do processador
- Boa parte dessa perda se deve ao sistema de memória
 - Mover dados leva muito mais tempo que as operações aritméticas e lógicas
- Para entender o por que, precisamos estudar um pouco sobre o funcionamento dos processadores modernos
 - Vamos assumir por enquanto um processador single core
 - Mas estas questões aparecem em qualquer computador paralelo

CPU/Memory performance



Slide 17

Computer architecture: a quantitative approach

By John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau

Gap Processor-DRAM

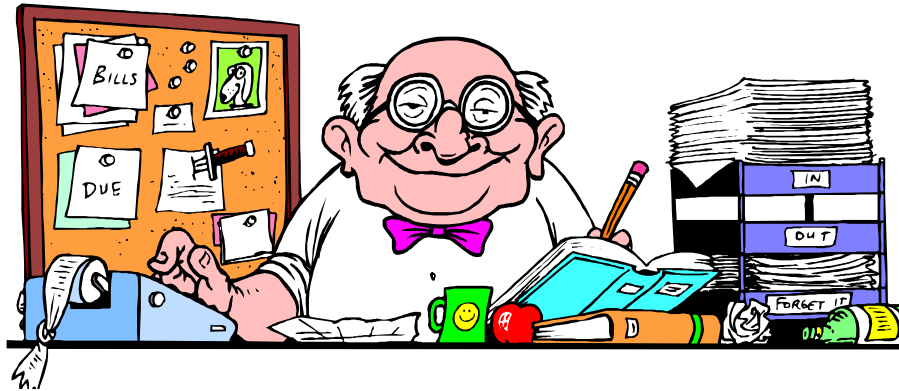
01/21/2016

CS267 - Lecture 2

Abordagens para tratar a latência de memória

- Eliminar operações de memória armazenando os valores em memória rápida (cache) e reusando-os
 - **Precisamos de localidade temporal no programa**
- Aproveitar a maior largura de banda trazendo um bloco da memória, armazenando-o em memória rápida (cache) e usando o bloco todo
 - **A largura de banda melhora mais rápido que a latência: 23% vs 7% por ano**
 - **Precisamos de localidade espacial no programa**
- Aproveitar a melhor largura de banda permitindo ao processador realizar várias leituras ao sistema de memória de uma vez
 - **concorrência no stream de instruções, ou seja, carregar um array inteiro, como em processadores vetoriais, ou prefetching**
- Sobrepor operações de computação e memória
 - **prefetching**

Caching Concept




- **Cache**: um repositório para cópias que podem ser acessadas mais rápido que o original
- Somente serve se:
 - O caso frequente é frequente suficiente
 - O caso infrequente não é caro demais
- Medida importante:
 - Tempo de acesso médio = $(\text{Hit Rate} \times \text{Hit Time}) + (\text{Miss Rate} \times \text{Miss Time})$

Note on Matrix Storage

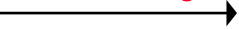
- A matrix is a 2-D array of elements, but memory addresses are “1-D”
- Conventions for matrix layout
 - by column, or “column major” (Fortran default); $A(i,j)$ at $A+i*j*n$
 - by row, or “row major” (C default) $A(i,j)$ at $A+i*n+j$
 - recursive (later)

Column major



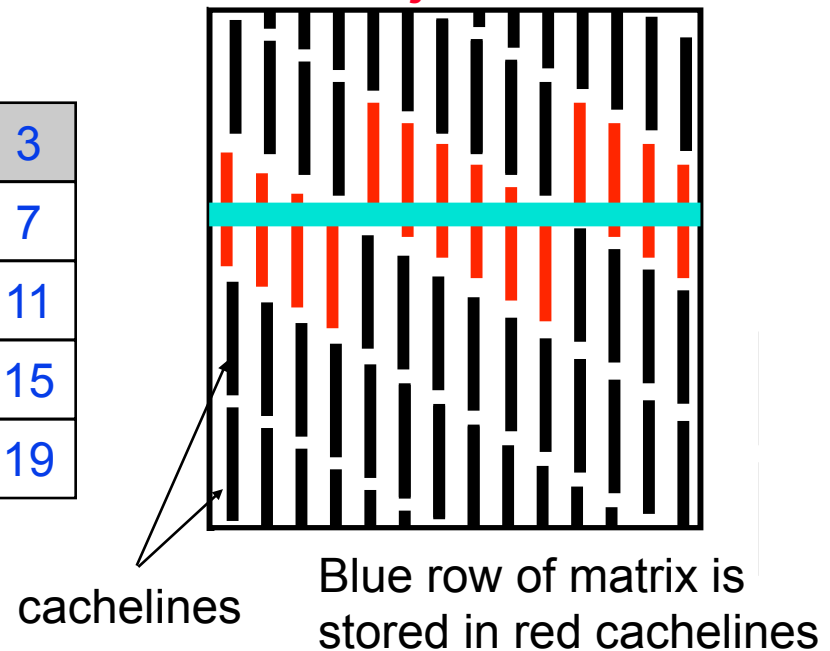
0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

Row major



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Column major matrix in memory



- Column major (for now)

(a) Version *ijk*

```

1  for (i = 0; i < n; i++)
2      for (j = 0; j < n; j++) {
3          sum = 0.0;
4          for (k = 0; k < n; k++)
5              sum += A[i][k]*B[k][j];
6          C[i][j] += sum;
7      }

```

code/mem/matmult/mm.c

(b) Version *jik*

```

1  for (j = 0; j < n; j++)
2      for (i = 0; i < n; i++) {
3          sum = 0.0;
4          for (k = 0; k < n; k++)
5              sum += A[i][k]*B[k][j];
6          C[i][j] += sum;
7      }

```

code/mem/matmult/mm.c

(c) Version *jki*

```

1  for (j = 0; j < n; j++)
2      for (k = 0; k < n; k++) {
3          r = B[k][j];
4          for (i = 0; i < n; i++)
5              C[i][j] += A[i][k]*r;
6      }

```

code/mem/matmult/mm.c

(d) Version *kji*

```

1  for (k = 0; k < n; k++)
2      for (j = 0; j < n; j++) {
3          r = B[k][j];
4          for (i = 0; i < n; i++)
5              C[i][j] += A[i][k]*r;
6      }

```

code/mem/matmult/mm.c

(e) Version *kij*

```

1  for (k = 0; k < n; k++)
2      for (i = 0; i < n; i++) {
3          r = A[i][k];
4          for (j = 0; j < n; j++)
5              C[i][j] += r*B[k][j];
6      }

```

code/mem/matmult/mm.c

(f) Version *ikj*

```

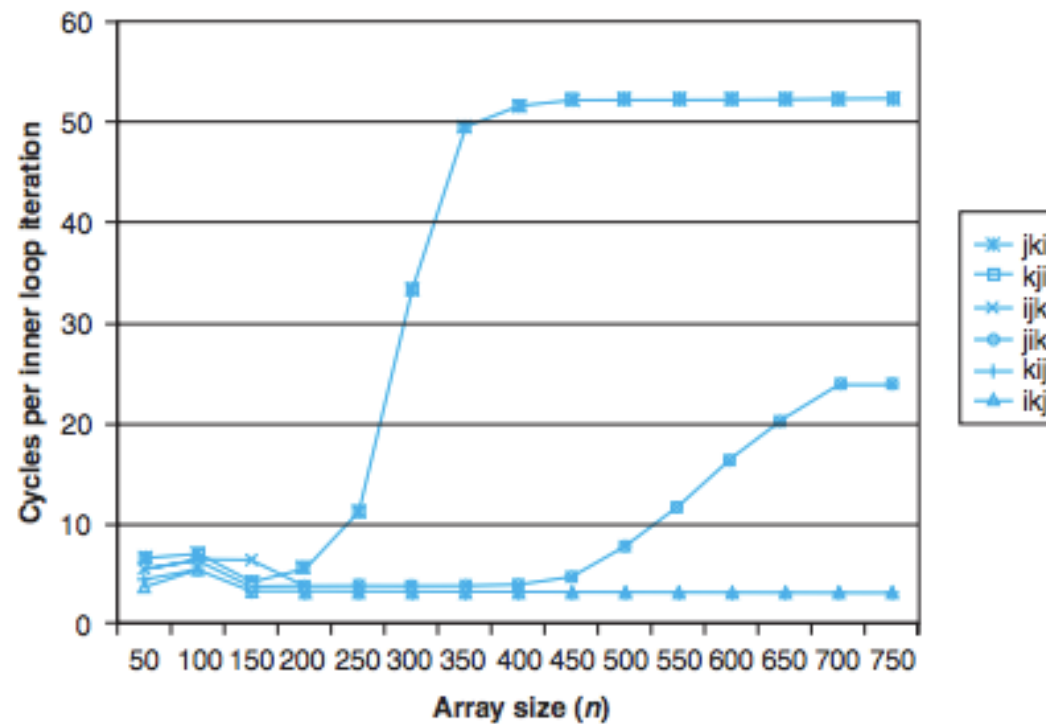
1  for (i = 0; i < n; i++)
2      for (k = 0; k < n; k++) {
3          r = A[i][k];
4          for (j = 0; j < n; j++)
5              C[i][j] += r*B[k][j];
6      }

```

code/mem/matmult/mm.c

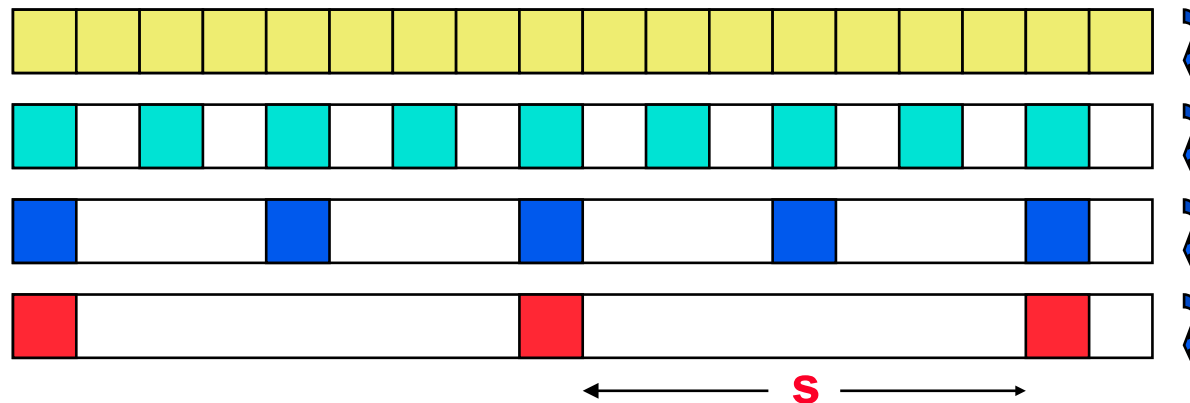
Desempenho da multiplicação de matrizes no Core I7

Matrix multiply version (class)	Loads per iter.	Stores per iter.	A misses per iter.	B misses per iter.	C misses per iter.	Total misses per iter.
<i>ijk</i> & <i>jik</i> (AB)	2	0	0.25	1.00	0.00	1.25
<i>jki</i> & <i>kji</i> (AC)	2	1	1.00	0.00	1.00	2.00
<i>kij</i> & <i>ikj</i> (BC)	2	1	0.00	0.25	0.25	0.50



Engenharia reversa: Estudo experimental de memória (Membench)

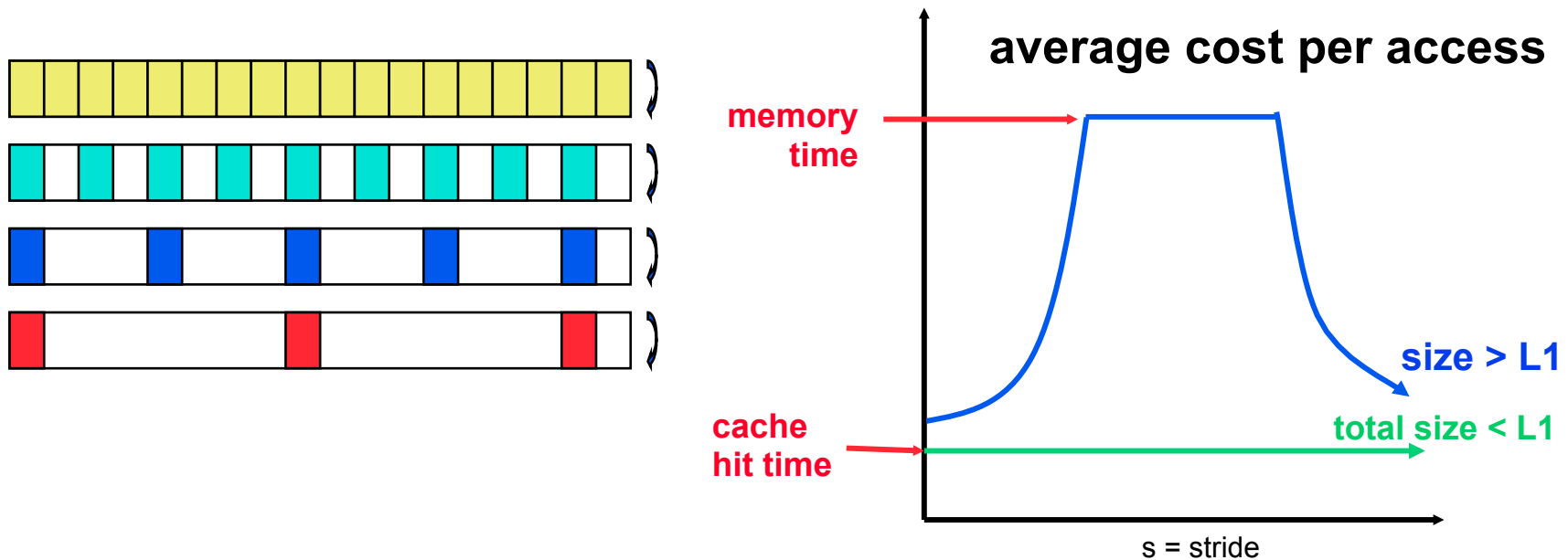
- Microbenchmark : desempenho do sistema de memória



- 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-1 **by s**
load A[i] from memory (4 Bytes)

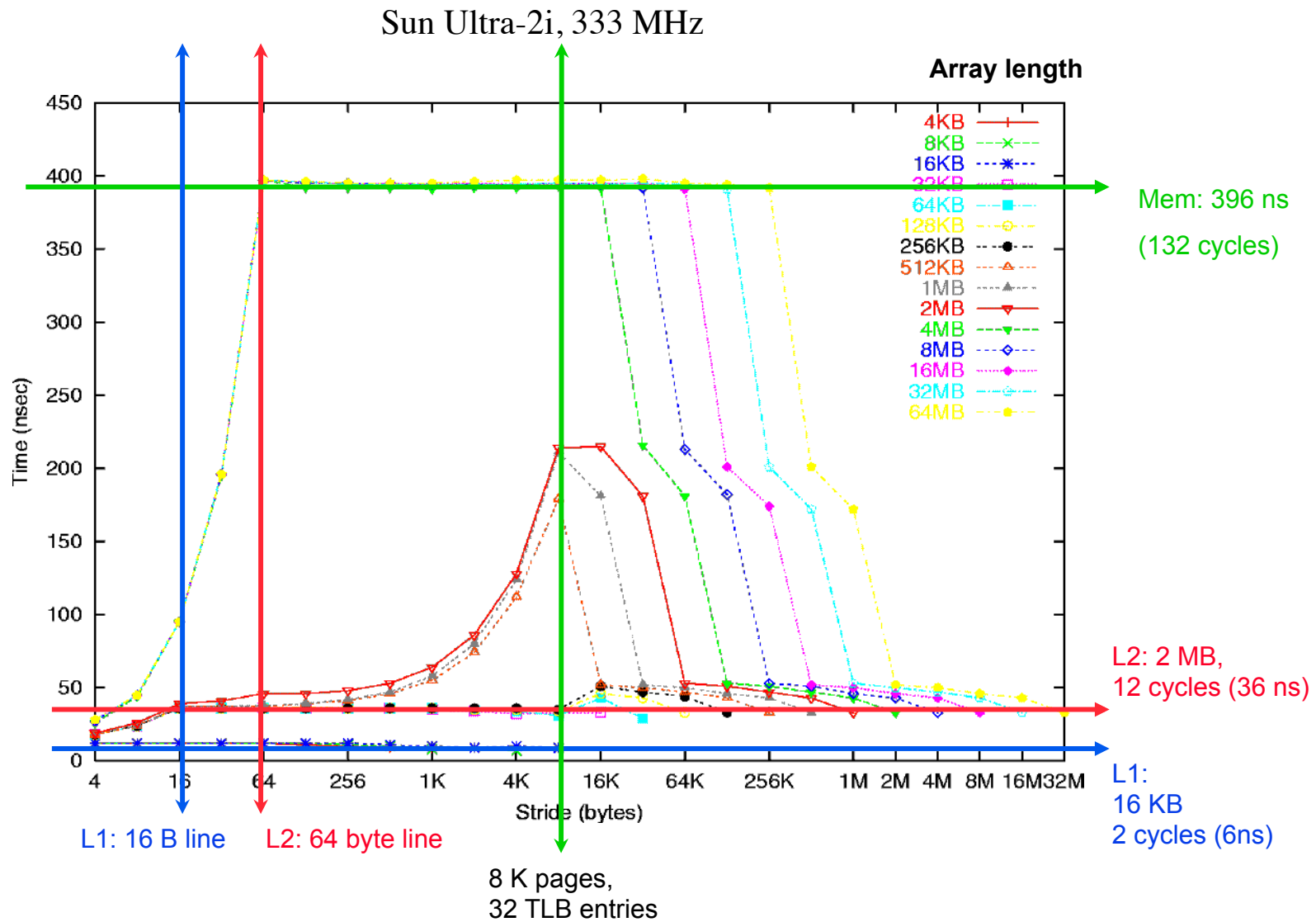
1 experimento

Engenharia reversa: Membench: 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

Engenharia reversa: Hierarquia de Mem na Sun Ultra-2i



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

Virtual Memory Review

- Goal: give illusion of a large memory
- Allow many processes to share single memory
- Strategy
 - Break physical memory up into blocks (*pages*)
 - Page might be in physical memory or on disk.
- Addresses:
 - Generated by lw/sw: *virtual*
 - Actual memory locations: *physical*

Memory access

- Load/store/PC computes a *virtual* address
- Need address translation
 - Convert virtual addr to physical addr
 - Use page table for lookup
 - Check virtual address:
 - If page is in memory, access memory with physical address
 - May also need to check access permissions
 - If page is not in memory, access disk
 - Page fault
 - Slow – so run another program while it's doing that
 - Do translation in hardware
 - Software translation would be too slow!

Handling a page fault

- Occurs during memory access clock cycle
- Handler must:
 - Find disk address from page table entry
 - Choose physical page to replace
 - If page dirty, write to disk first
 - Read referenced page from disk into physical page

TLB: Translation Lookaside Buffer

- Address translation has a high degree of locality
 - If page accessed once, highly likely to be accessed again soon.
 - So, cache a few frequently used page table entries
- TLB = hardware cache for the page table
 - Make translation faster
 - Small, frequently fully-associative
- TLB entries contain
 - Valid bit
 - Other housekeeping bits
 - Tag = virtual page number
 - Data = Physical page number
- Misses handled in hardware (dedicated FSM) or software (OS code reads page table)



TLB Misses

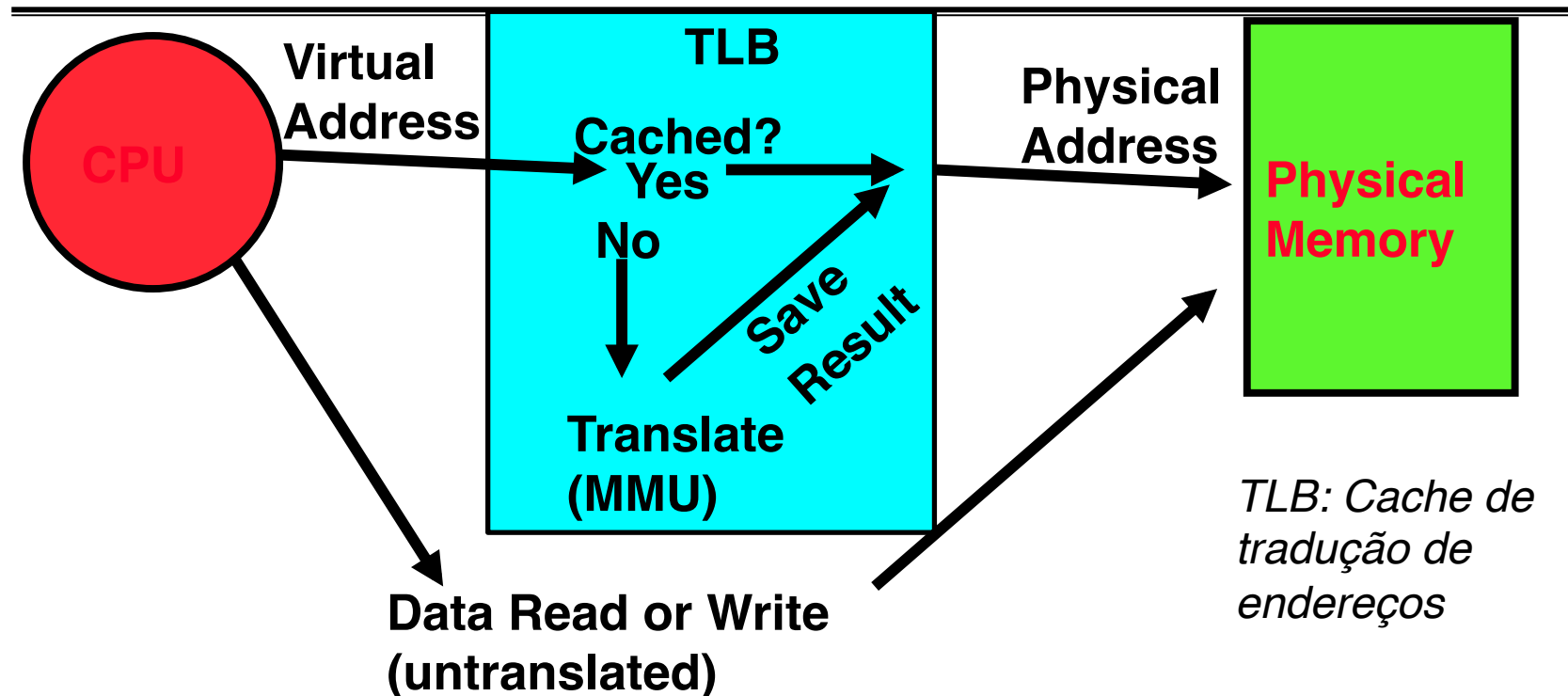
- TLB miss means one of two things
 - Page is present in memory, need to create the missing mapping in the TLB
 - Page is not present in memory (page fault), need to transfer control to OS to deal with it.
 - Need to generate an exception
 - Copy page table entry to TLB – use appropriate replacement algorithm if you need to evict an entry from TLB.



Optimizations

- Make the common case fast
- Speed up TLB hit + L1 Cache hit
 - Do TLB lookup and cache lookup in parallel
 - Possible if cache index is independent of virtual address translation
 - Have cache indexed by virtual addresses

Caching Applied to Address Translation



- Question is one of page locality: does it exist?
 - Instruction accesses spend a lot of time on the same page (since accesses sequential)
 - Stack accesses have definite locality of reference
 - Data accesses have less page locality, but still some...
- Can we have a TLB hierarchy?
 - Sure: multiple levels at different sizes/speeds

Memory Hierarchy on a Power3 (Seaborg)

Power3, 375 MHz

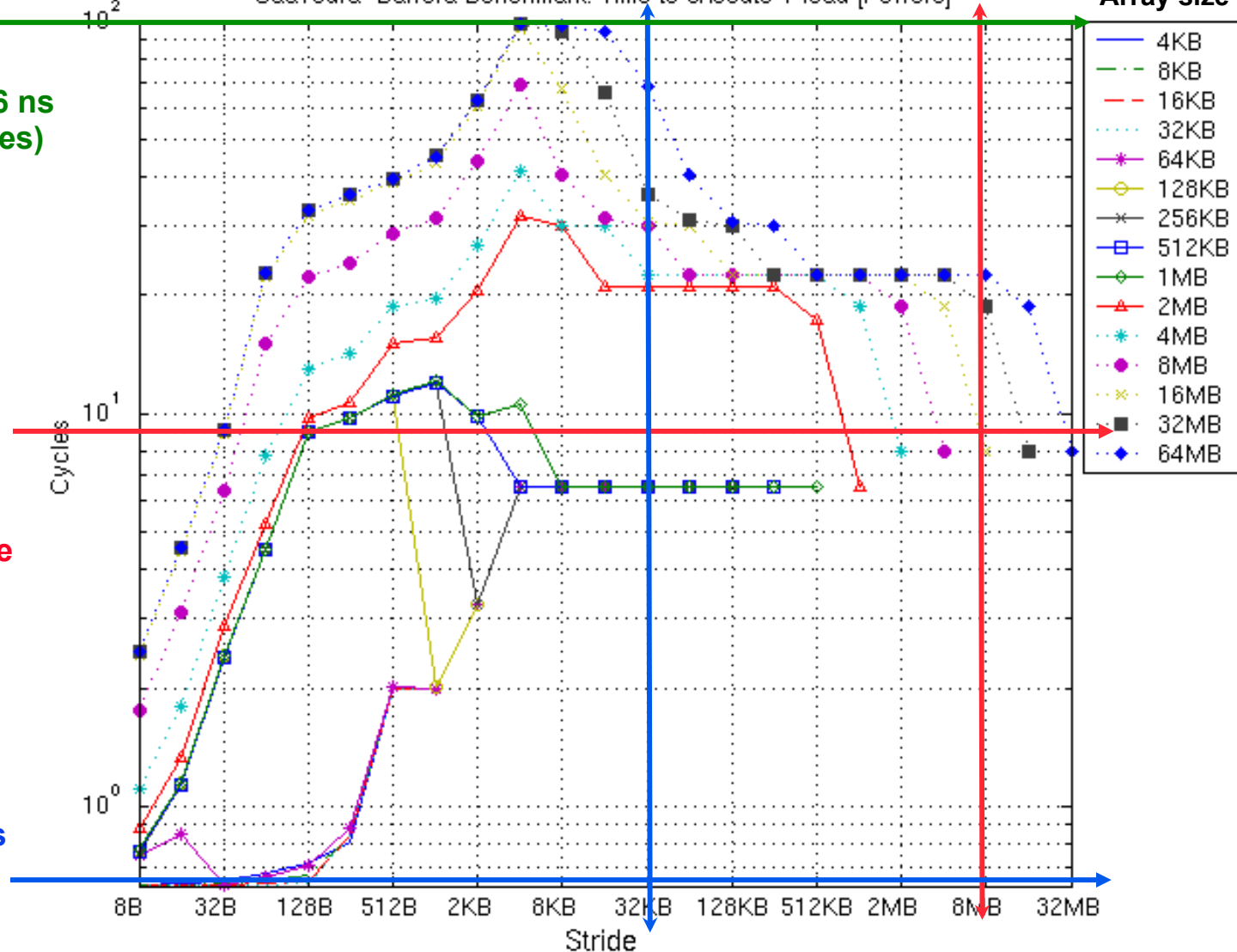
Saavedra-Barrera Benchmark: Time to execute 1 load [Power3]

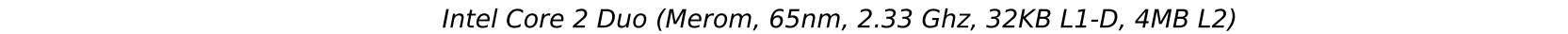
Array size

Mem: 396 ns
(132 cycles)

L2: 8 MB
128 B line
9 cycles

L1: 32 KB
128B line
.5-2 cycles





Idealized Uniprocessor Model

- Processor names bytes, words, etc. in its address space
 - These represent integers, floats, pointers, arrays, etc.
- Operations include
 - Read and write into very fast memory called registers
 - Arithmetic and other logical operations on registers
- Order specified by program
 - Read returns the most recently written data
 - Compiler and architecture translate high level expressions into “obvious” lower level instructions

$A = B + C \Rightarrow$

Read address(B) to R1
Read address(C) to R2
 $R3 = R1 + R2$
Write R3 to Address(A)

- Hardware executes instructions in order specified by compiler
- *Idealized Cost*
 - Each operation has roughly the same cost
(read, write, add, multiply, etc.)

Uniprocessors in the Real World

- Real 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 processor

*In theory there is no difference between theory and practice.
But in practice there is. - Yogi Berra*

SIMD: Single Instruction, Multiple Data

- Scalar processing

- traditional mode
- one operation produces one result



+

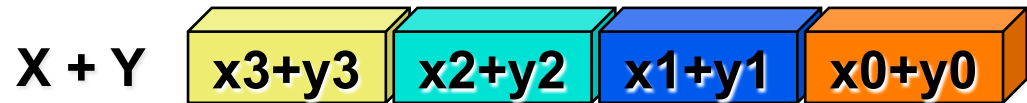
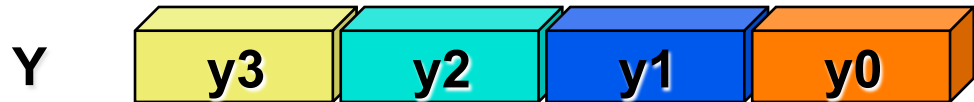


- SIMD processing

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



+



Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

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** or **tiling**
 - Idea: used divide-and-conquer to define a problem that fits in register/L1-cache/L2-cache

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
 - It works with the mix of instructions inside an inner loop or other block of code
- 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

- Idealized and actual costs in modern processors
- Memory hierarchies
 - Use of microbenchmarks to characterized performance
- Parallelism within single processors
- Case study: Matrix Multiplication
 - Use of performance models to understand performance
 - Attainable lower bounds on communication
 - Simple cache model
 - Warm-up: Matrix-vector multiplication
 - Naïve vs optimized Matrix-Matrix Multiply
 - Minimizing data movement
 - Beating $O(n^3)$ operations
 - Practical optimizations

Optimizing for the serial processors

- Scaled speedup: operate near the memory boundary.
- Memory systems on modern processors are complicated.
- The performance of a simple program can depend on the details of the micro-architecture.
- Today we will study matrix multiplication optimizations
 - An important kernel in some scientific problems
 - Closely related to other algorithms, e.g., transitive closure on a graph using Floyd-Warshall (is there a path between arbitrary vertices)
 - Optimization ideas can be used in other problems
 - The best case for optimization payoffs
 - The most well-studied algorithm in high performance computing

Why Matrix Multiplication?

- An important kernel in many problems
 - Appears in many linear algebra algorithms
 - Bottleneck for dense linear algebra, including Top500
 - One of the 7 dwarfs / 13 motifs of parallel computing
 - Closely related to other algorithms, e.g., transitive closure on a graph using Floyd-Warshall
- Optimization ideas can be used in other problems
- The best case for optimization payoffs
- The most-studied algorithm in high performance computing

What do commercial and CSE applications have in common?

Motif/Dwarf: Common Computational Methods (Red Hot → Blue Cool)

	Embed	SPEC	DB	Games	ML	HPC	Health	Image	Speech	Music	Browser
1 Finite State Mach.	Red	Red	Red	Yellow	Yellow	Light Blue	Light Blue	Light Blue	Light Blue	Light Blue	Red
2 Combinational	Red	Light Blue	Light Green	Light Blue	Light Green	Light Blue	Light Blue	Light Blue	Light Blue	Light Blue	Red
3 Graph Traversal	Red	Yellow	Yellow	Yellow	Red	Light Blue	Red	Light Blue	Red	Light Green	Light Green
4 Structured Grid	Red	Red	Light Blue	Yellow	Light Blue	Red	Light Blue	Red	Light Blue	Light Blue	Light Blue
5 Dense Matrix	Red	Red	Yellow	Red	Red	Red	Light Blue	Red	Red	Red	Light Blue
6 Sparse Matrix	Yellow	Yellow	Light Blue	Red	Red	Red	Red	Light Blue	Light Blue	Red	Light Blue
7 Spectral (FFT)	Yellow	Light Blue	Light Blue	Yellow	Yellow	Red	Light Blue	Light Green	Red	Red	Red
8 Dynamic Prog	Yellow	Light Blue	Red	Light Blue	Red	Light Blue	Light Blue	Light Blue	Yellow	Light Blue	Red
9 N-Body	Light Blue	Yellow	Light Blue	Yellow	Light Blue	Red	Light Green	Light Blue	Light Blue	Light Blue	Light Blue
10 MapReduce	Light Blue	Light Green	Red	Light Blue	Red	Red	Red	Red	Yellow	Red	Yellow
11 Backtrack/ B&B	Light Blue	Light Blue	Yellow	Light Blue	Red	Light Blue	Light Blue	Light Blue	Light Blue	Yellow	Light Blue
12 Graphical Models	Light Blue	Light Blue	Yellow	Light Blue	Red	Light Blue	Light Blue	Light Blue	Light Blue	Red	Light Blue
13 Unstructured Grid	Light Blue	Light Blue	Light Blue	Yellow	Yellow	Red	Red	Light Blue	Light Blue	Red	Light Blue

Um modelo simples para otimizar o algoritmo

- Assume just 2 levels in the 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 $\ll t_m$
 - $q = f / m$ average number of flops per slow memory access
- Minimum possible time = $f * t_f$ when all data in fast memory
- Actual time
 - $f * t_f + m * t_m = f * t_f * (1 + \frac{t_m}{t_f} * 1/q)$
- 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

Computational Intensity: Key to algorithm efficiency

Machine Balance: Key to machine efficiency

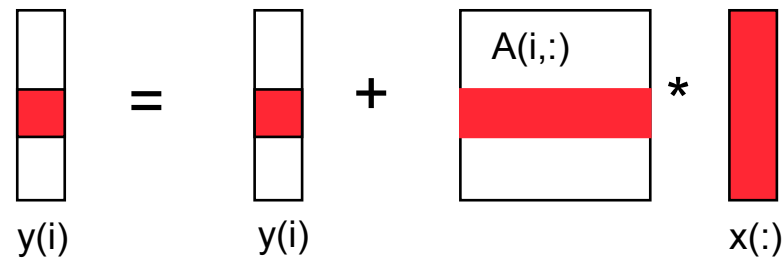
Warm up: Matrix-vector multiplication

```
{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

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

Modeling Matrix-Vector Multiplication

- Compute time for $n \times n = 1000 \times 1000$ matrix
- Time
 - $f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/q)$
 - $= 2 * n^2 * t_f * (1 + t_m/t_f * 1/2)$
- For t_f and t_m , using data from R. Vuduc's PhD (pp 351-3)
 - <http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf>
 - For t_m use minimum-memory-latency / words-per-cache-line

	Clock	Peak	Mem Lat (Min,Max)		Linesize	t _m /t _f
	MHz	Mflop/s	cycles		Bytes	
Ultra 2i	333	667	38	66	16	24.8
Ultra 3	900	1800	28	200	32	14.0
Pentium 3	500	500	25	60	32	6.3
Pentium3M	800	800	40	60	32	10.0
Power3	375	1500	35	139	128	8.8
Power4	1300	5200	60	10000	128	15.0
Itanium1	800	3200	36	85	32	36.0
Itanium2	900	3600	11	60	64	5.5

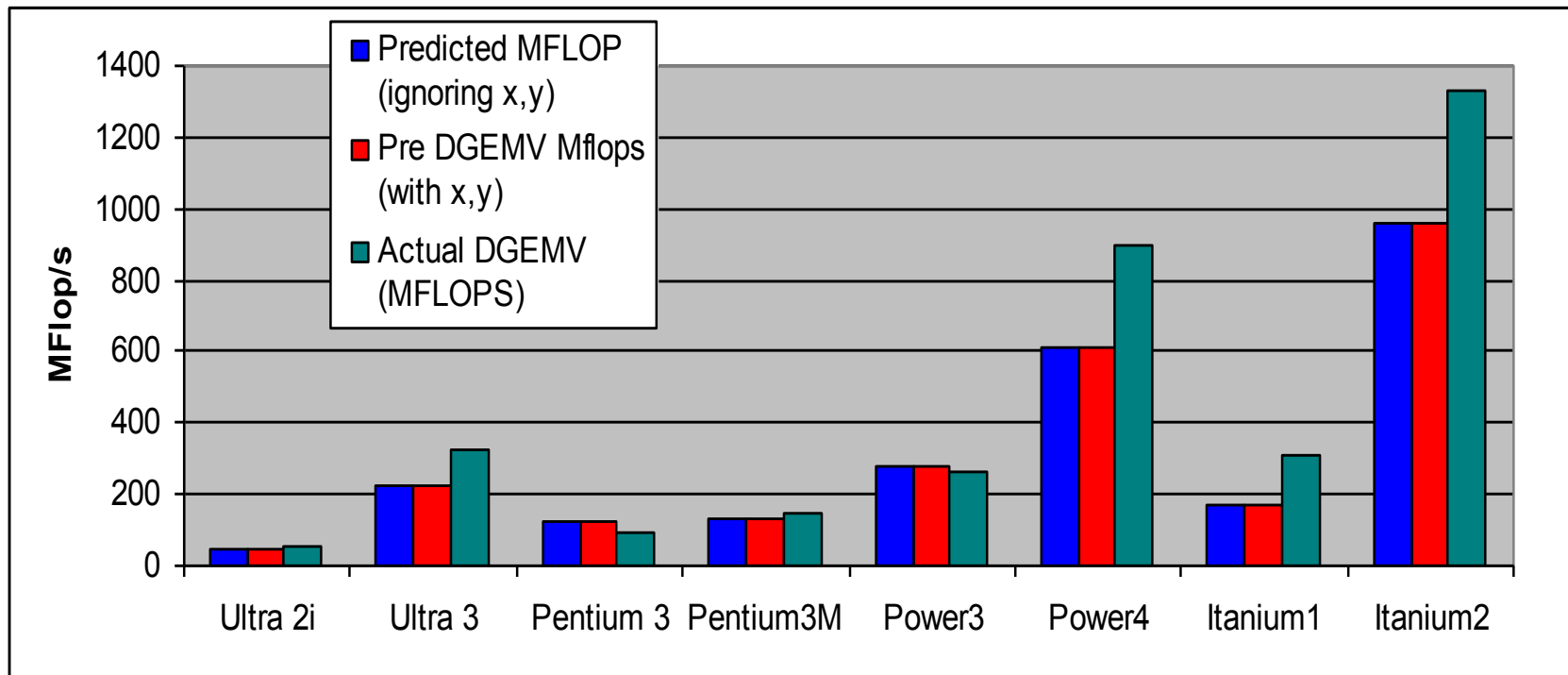
*machine
balance
(q must
be at least
this for
½ peak
speed)*

Simplifying Assumptions

- What simplifying assumptions did we make in this analysis?
 - Ignored parallelism in processor between memory and arithmetic within the processor
 - Sometimes drop arithmetic term in this type of analysis
 - Assumed fast memory was large enough to hold three vectors
 - Reasonable if we are talking about any level of cache
 - Not if we are talking about registers (~32 words)
 - Assumed the cost of a fast memory access is 0
 - Reasonable if we are talking about registers
 - Not necessarily if we are talking about cache (1-2 cycles for L1)
 - Memory latency is constant
- Could simplify even further by ignoring memory operations in X and Y vectors
 - Mflop rate/element = $2 / (2 * t_f + t_m)$

Validating the Model

- How well does the model predict actual performance?
 - Actual DGEMV: Most highly optimized code for the platform
- Model sufficient to compare across machines
- But under-predicting on most recent ones due to latency estimate

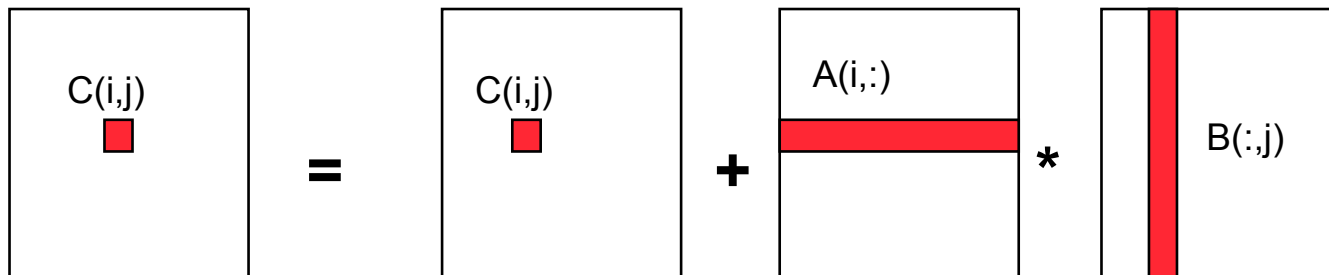


Naïve Matrix Multiply

```
{implements  $C = C + A * B$ }  
for i = 1 to n  
  for j = 1 to n  
    for k = 1 to n  
       $C(i,j) = C(i,j) + A(i,k) * B(k,j)$ 
```

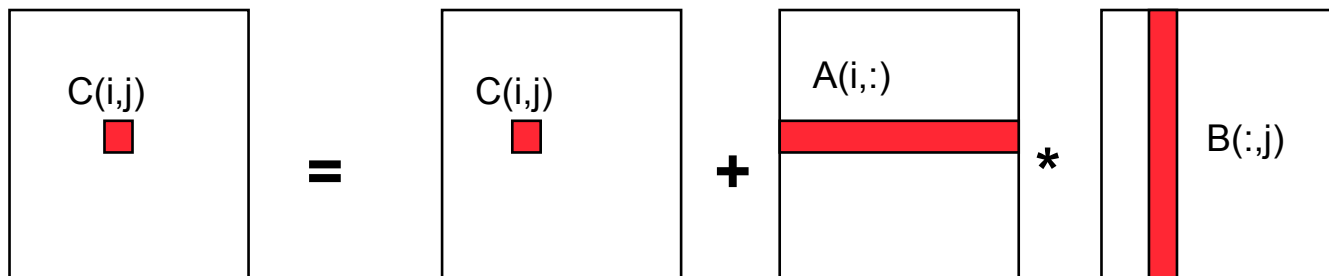
Algorithm has $2 * n^3 = O(n^3)$ Flops and
operates on $3 * n^2$ words of memory

q potentially as large as $2 * n^3 / 3 * n^2 = O(n)$



Naïve Matrix Multiply

```
{implements  $C = C + A*B$ }  
for i = 1 to n  
  {read row i of A into fast memory}  
  for j = 1 to n  
    {read  $C(i,j)$  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  $C(i,j)$  back to slow memory}
```



Naïve Matrix Multiply

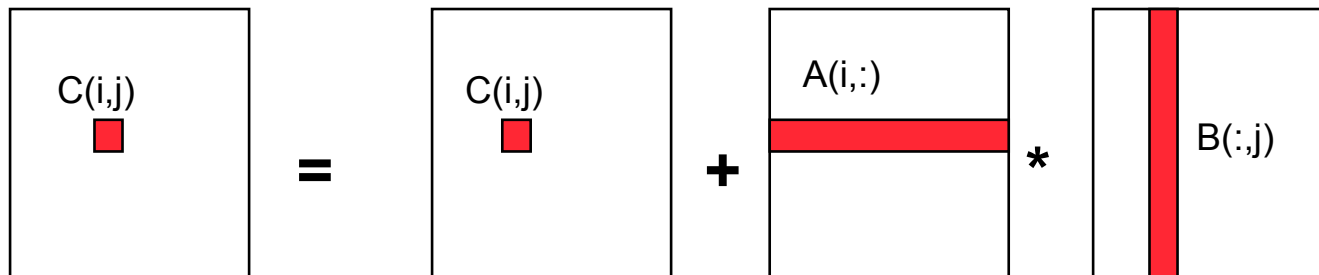
Number of slow memory references on unblocked matrix multiply

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

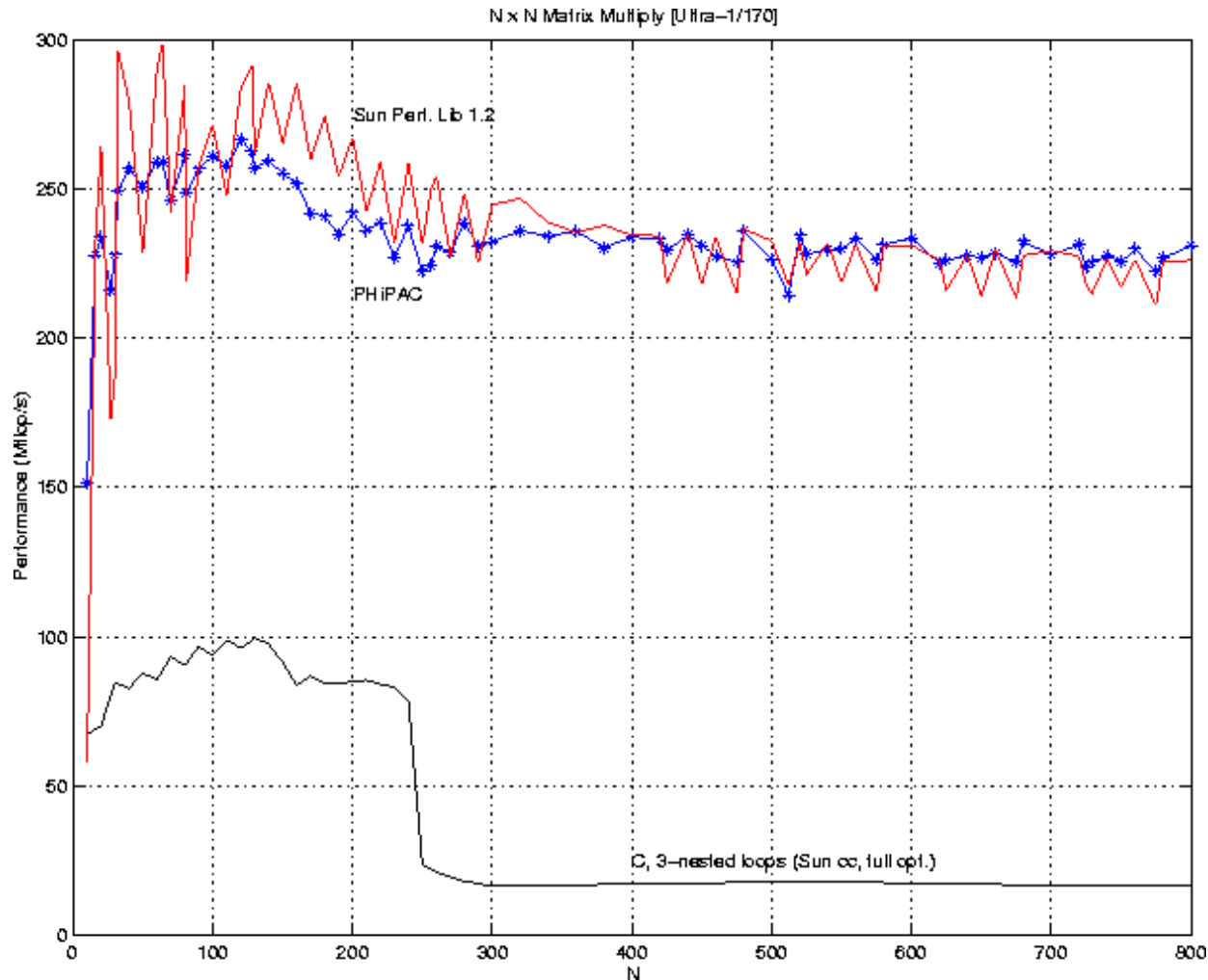
$$\text{So } q = f / m = 2n^3 / (n^3 + 3n^2)$$

≈ 2 for large n , no improvement over matrix-vector multiply

Inner two loops are just matrix-vector multiply, of row i of A times B
Similar for any other order of 3 loops

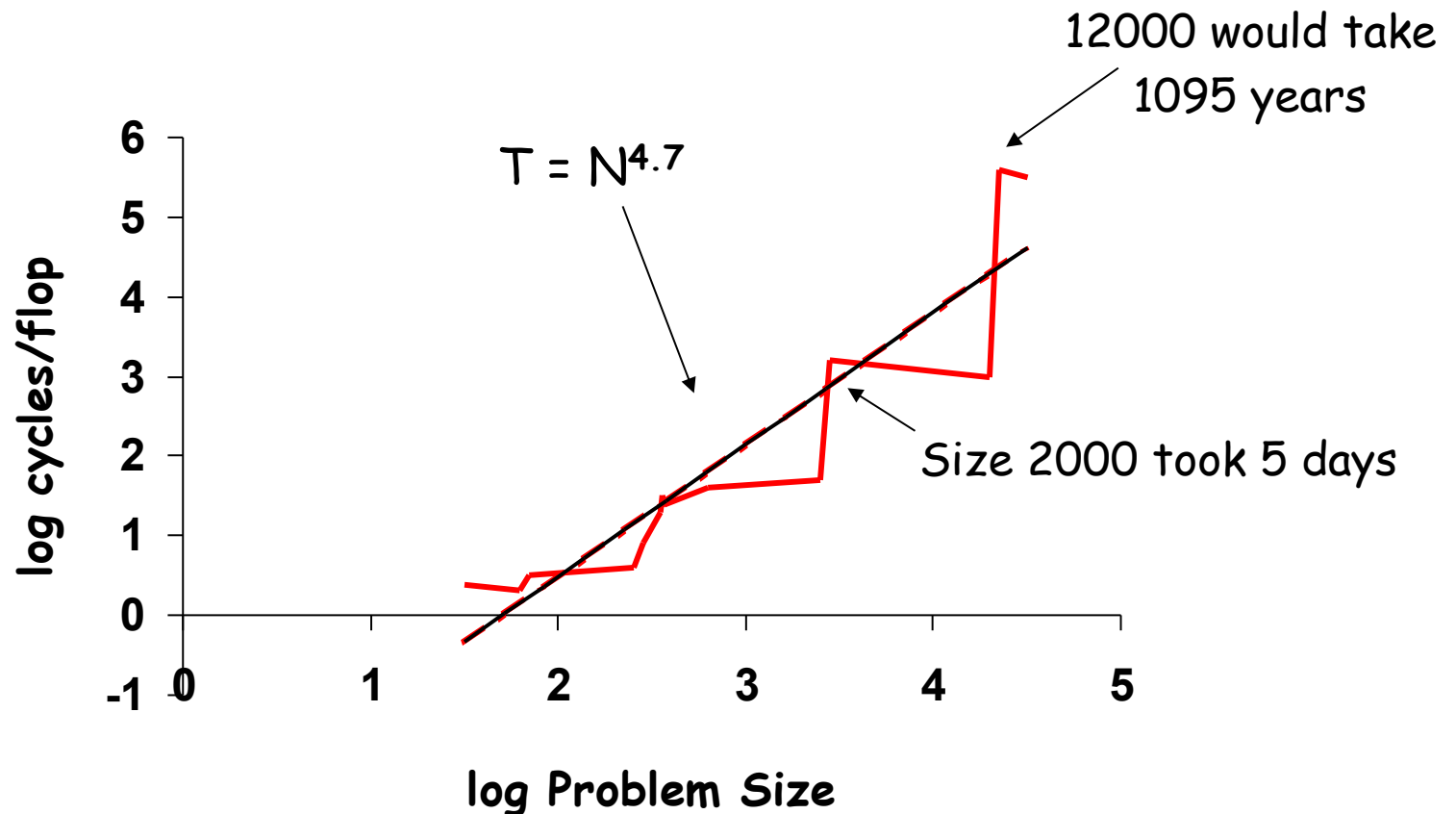


Matrix-multiply, optimized several ways



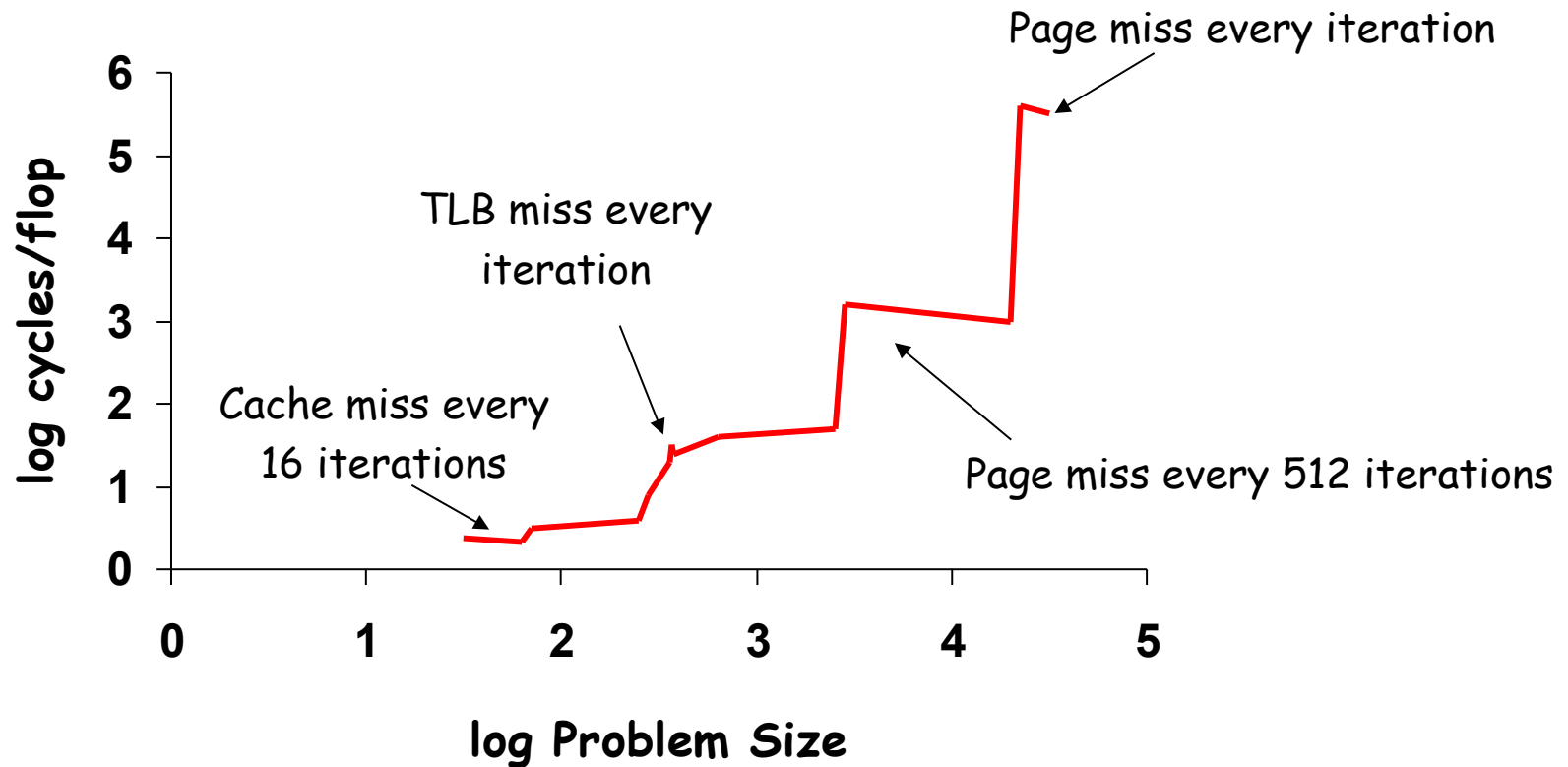
Speed of n-by-n matrix multiply on Sun Ultra-1/170, peak = 330 MFlops

Naïve Matrix Multiply on RS/6000



$O(N^3)$ performance would have constant cycles/flop
Performance looks like $O(N^{4.7})$

Naïve Matrix Multiply on RS/6000



“Naïve” Matrix Multiply

Number of slow memory references on unblocked matrix multiply

$m = n^3$ read each column of B n times

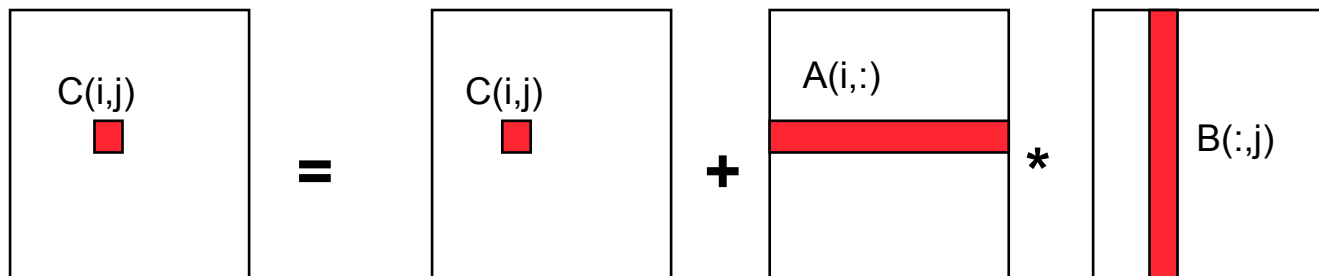
+ n^2 read each row of A once

+ $2n^2$ read and write each element of C once

= $n^3 + 3n^2$

So $q = f / m = 2n^3 / (n^3 + 3n^2)$

~ 2 for large n , no improvement over matrix-vector multiply



Blocked (Tiled) Matrix Multiply

Consider A,B,C to be 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}

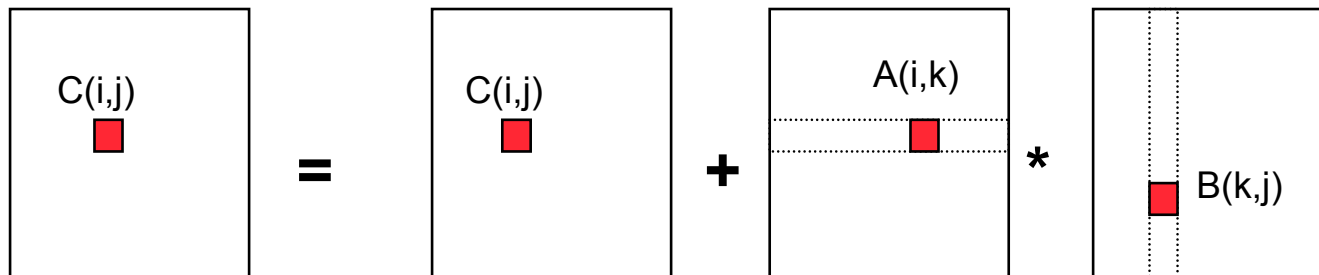
for k = 1 to N

{read block A(i,k) into fast memory}

{read block B(k,j) into fast memory}

$C(i,j) = C(i,j) + A(i,k) * B(k,j)$ {do a matrix multiply on blocks}

{write block C(i,j) back to slow memory}



Blocked (Tiled) Matrix Multiply

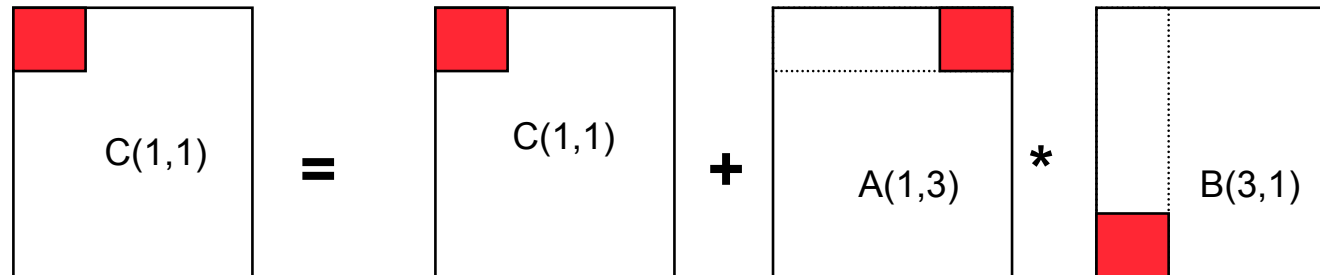
The diagram illustrates the first step of blocked matrix multiplication. It shows the equation $C(1,1) = C(1,1) + A(1,1) * B(1,1)$. Each matrix is represented by a square with a red top-left tile. The matrix $A(1,1)$ has a horizontal dotted line, and the matrix $B(1,1)$ has a vertical dotted line.

Blocked (Tiled) Matrix Multiply

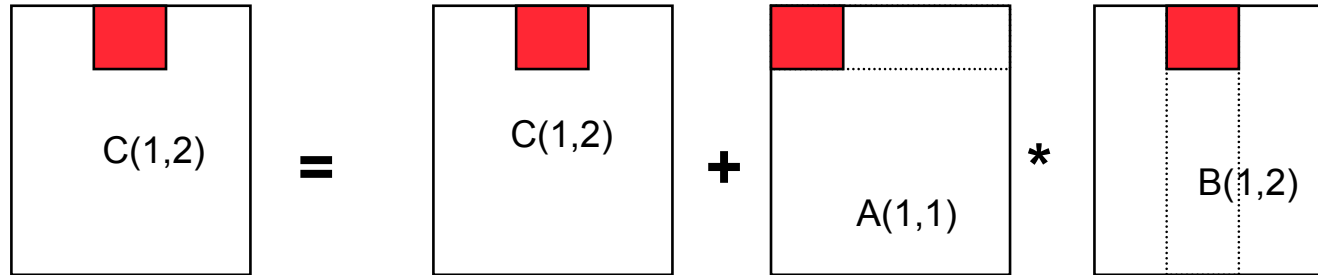
The diagram illustrates a blocked matrix multiplication operation. It consists of four square blocks representing matrices, connected by mathematical operators. The first block is a square with a red square in the top-left corner and the label $C(1,1)$ in the center. This is followed by an equals sign. The second block is identical to the first, also labeled $C(1,1)$. This is followed by a plus sign. The third block is a square with a red square in the top-right corner, a horizontal dotted line below the red square, and the label $A(1,2)$ in the center. This is followed by an asterisk. The fourth block is a square with a red square in the middle-left position, a vertical dotted line to the left of the red square, and the label $B(2,1)$ in the center.

$$C(1,1) = C(1,1) + A(1,2) * B(2,1)$$

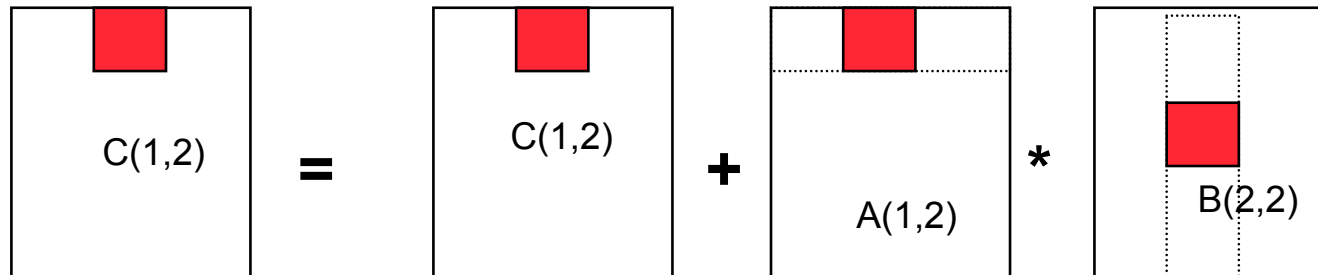
Blocked (Tiled) Matrix Multiply



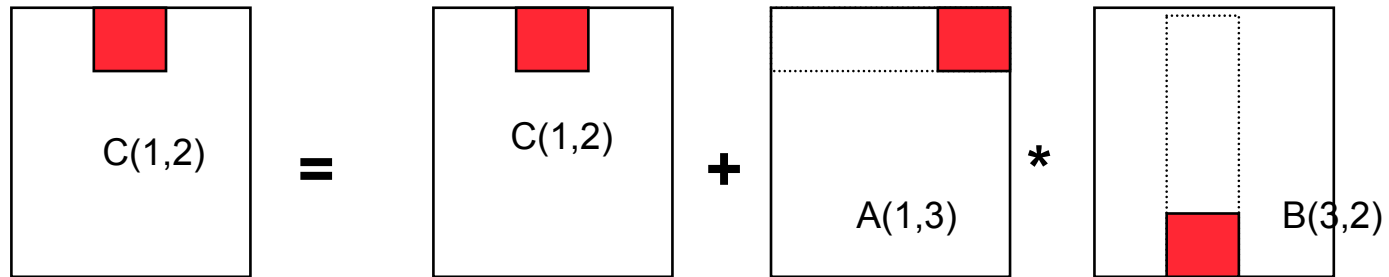
Blocked (Tiled) Matrix Multiply



Blocked (Tiled) Matrix Multiply



Blocked (Tiled) Matrix Multiply



Blocked (Tiled) Matrix Multiply

Recall:

m is amount memory traffic between slow and fast memory

matrix has $n \times n$ elements, and $N \times N$ blocks each of size $b \times b$

f is number of floating point operations, $2n^3$ for this problem

$q = f / m$ is our measure of algorithm efficiency in the memory system

So:

$$\begin{aligned} m &= N * n^2 && \text{read each block of B } N^3 \text{ times } (N^3 * n/N * n/N) \\ &+ N * n^2 && \text{read each block of A } N^3 \text{ times} \\ &+ 2n^2 && \text{read and write each block of C once} \\ &= (2N + 2) * n^2 \end{aligned}$$

So computational intensity $q = f / m = 2n^3 / ((2N + 2) * n^2)$

$$\sim n / N = b \text{ for large } n$$

So we can improve performance by increasing the blocksize b

Can be much faster than matrix-vector multiply ($q=2$)

Linux

**/sys/devices/system/cpu/cpu0/
cache/**

**coherency_line_size (bytes)
level
number_of_sets
physical_line_partition
shared_cpu_list
shared_cpu_map
size
type
ways_of_associativity**

man line

**[http://manpages.ubuntu.com/
manpages/precise/line.8.html](http://manpages.ubuntu.com/manpages/precise/line.8.html)**

/proc/cpuinfo

lstopo-no-graphics (veja hwloc)

sudo dmidecode -t cache

- `lscpu | grep cache`

- L1d **cache:** 32K
- L1i **cache:** 32K
- L2 **cache:** 256K
- L3 **cache:** 8192K

- `/sys/devices/system/cpu/cpu0/cache/index1/coherency_line_size`

- 64 bytes

- `valgrind --tool=cachegrind ./executavel`

Usando o modelo para entender as máquinas

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}

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

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

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

- These sizes are reasonable for L1 cache, but not for register sets
- Note: analysis assumes it is possible to schedule the instructions perfectly

		required
	t_m/t_f	KB
Ultra 2i	24.8186	14.8
Ultra 3	14	4.7
Pentium 3	6.25	0.9
Pentium3M	10	2.4
Power3	8.75	1.8
Power4	15	5.4
Itanium1	36	31.1
Itanium2	5.5	0.7

Blocked (Tiled) Matrix Multiply

Matriz A : array $q \times q$ de blocos $A_{i,j}$ ($0 \leq i, j < q$) tal que cada bloco é uma submatriz $(n/q) \times (n/q)$

procedure BLOCK_MAT_MULT(A, B, C)

begin

 for $i:=0$ to $q-1$ do

 for $j:=0$ to $q-1$ do

 begin

 Inicializar os elementos de $C_{i,j}$ com 0

 for $k:=0$ to $q-1$ do

$C_{i,j} := C_{i,j} + A_{i,k} * B_{k,j};$

 endfor;

 end BLOCK_MAT_MULT

Limits to Optimizing Matrix Multiply

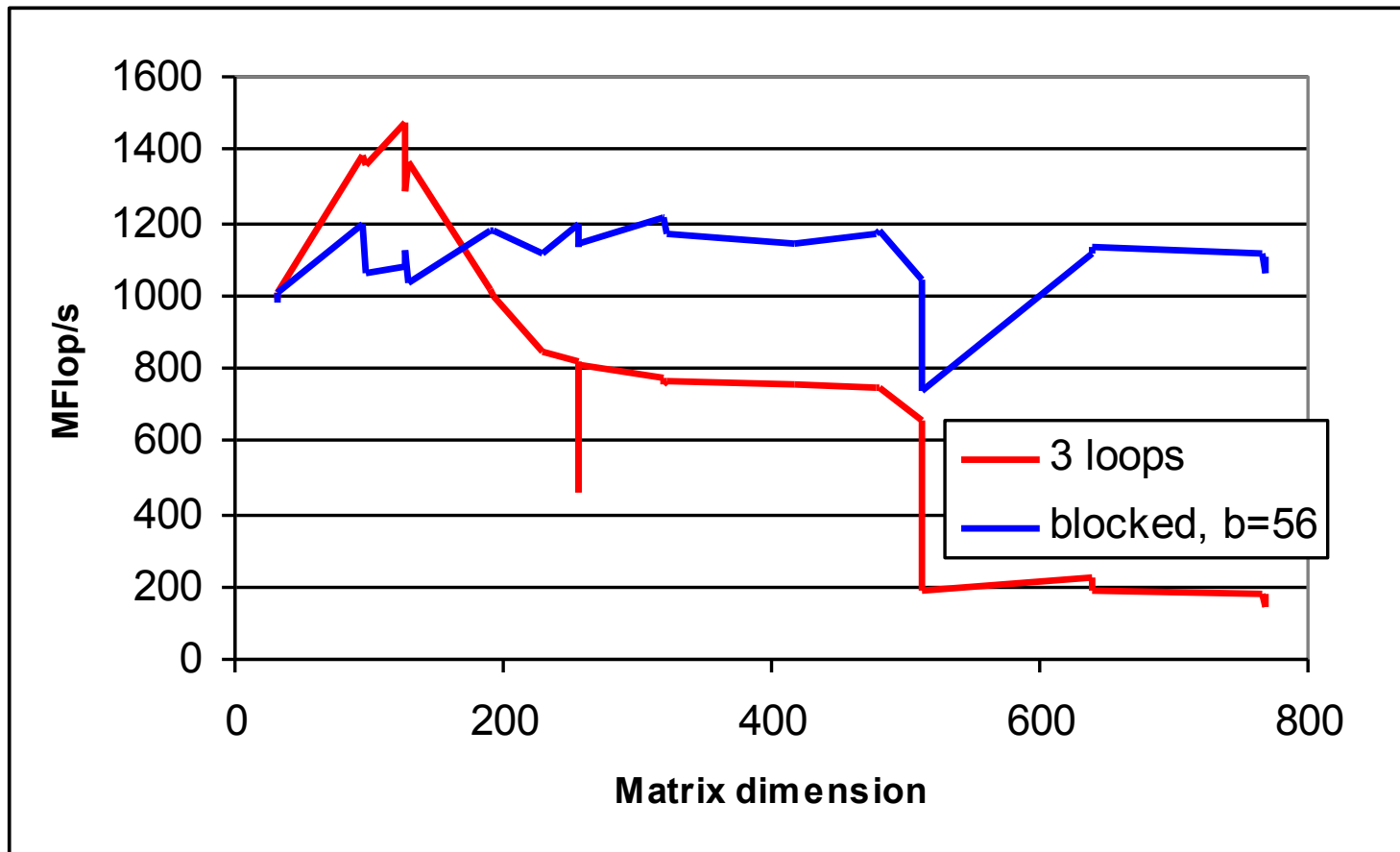
- The blocked algorithm changes the order in which values are accumulated into each $C[i,j]$ by applying associativity
- The previous analysis showed that the blocked algorithm has computational intensity:

$$q \approx b \leq \sqrt{M_{\text{fast}}/3}$$

- There is a lower bound result that says we cannot do any better than this (using only algebraic associativity)
- Theorem (Hong & Kung, 1981): Any reorganization of this algorithm (that uses only algebraic associativity) is limited to $q = O(\sqrt{M_{\text{fast}}})$

Tiling Alone Might Not Be Enough

- Naïve and a “naïvely tiled” code on Itanium 2
 - Searched all block sizes to find best, $b=56$
 - Starting point for next homework



Limits to Optimizing Matrix Multiply

- The blocked algorithm changes the order in which values are accumulated into each $C[i,j]$ by applying associativity
- The previous analysis showed that the blocked algorithm has computational intensity:

$$q \approx b \leq \sqrt{M_{\text{fast}}/3}$$

- There is a lower bound result that says we cannot do any better than this (using only algebraic associativity)
- Theorem (Hong & Kung, 1981): Any reorganization of this algorithm (that uses only algebraic associativity) is limited to $q = O(\sqrt{M_{\text{fast}}})$

Strassen's Matrix Multiply

- The traditional algorithm (with or without tiling) has $O(n^3)$ flops
- 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

$$\text{Let } M = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$\text{Let } p_1 = (a_{12} - a_{22}) * (b_{21} + b_{22})$$

$$p_5 = a_{11} * (b_{12} - b_{22})$$

$$p_2 = (a_{11} + a_{22}) * (b_{11} + b_{22})$$

$$p_6 = a_{22} * (b_{21} - b_{11})$$

$$p_3 = (a_{11} - a_{21}) * (b_{11} + b_{12})$$

$$p_7 = (a_{21} + a_{22}) * b_{11}$$

$$p_4 = (a_{11} + a_{12}) * b_{22}$$

$$\text{Then } m_{11} = p_1 + p_2 - p_4 + p_6$$

$$m_{12} = p_4 + p_5$$

$$m_{21} = p_6 + p_7$$

$$m_{22} = p_2 - p_3 + p_5 - p_7$$

Extends to nxn by divide&conquer

Strassen (continued)

$$\begin{aligned} T(n) &= \text{Cost of multiplying } n \times n \text{ matrices} \\ &= 7 * T(n/2) + 18 * (n/2)^2 \\ &= O(n^{\log_2 7}) \\ &= O(n^{2.81}) \end{aligned}$$

- Asymptotically faster
 - Several times faster for large n in practice
 - Cross-over depends on machine
 - “Tuning Strassen's Matrix Multiplication for Memory Efficiency”, M. S. Thottethodi, S. Chatterjee, and A. Lebeck, in Proceedings of Supercomputing '98
- Possible to extend communication lower bound to Strassen
 - #words moved between fast and slow memory
 $= \Omega(n^{\log_2 7} / M^{(\log_2 7)/2 - 1}) \sim \Omega(n^{2.81} / M^{0.4})$
(Ballard, D., Holtz, Schwartz, 2011, **SPAA Best Paper Prize**)
 - Attainable too, more on parallel version later

Other Fast Matrix Multiplication Algorithms

- World's record was $O(n^{2.37548\dots})$
 - Coppersmith & Winograd, 1987
- New Record! 2.37548 reduced to 2.37293
 - Virginia Vassilevska Williams, UC Berkeley & Stanford, 2011
- Newer Record! 2.37293 reduced to 2.37286
 - Francois Le Gall, 2014
- Lower bound on #words moved can be extended to (some) of these algorithms (2015 thesis of Jacob Scott)
- Possibility of $O(n^{2+\epsilon})$ algorithm!
 - Cohn, Umans, Kleinberg, 2003
- Can show they all can be made numerically stable
 - D., Dumitriu, Holtz, Kleinberg, 2007
- Can do rest of linear algebra (solve $Ax=b$, $Ax=\lambda x$, etc) as fast, and numerically stably
 - D., Dumitriu, Holtz, 2008
- Fast methods (besides Strassen) may need unrealistically large n

Tuning Code in Practice

- Tuning code can be tedious
 - Lots of code variations to try besides blocking
 - Machine hardware performance hard to predict
 - Compiler behavior hard to predict
- Response: “Autotuning”
 - Let computer generate large set of possible code variations, and search them for the fastest ones
 - Used with CS267 homework assignment in mid 1990s
 - PHiPAC, leading to ATLAS, incorporated in Matlab
 - We still use the same assignment
 - We (and others) are extending autotuning to other dwarfs / motifs, eg FFT
 - Sometimes all done “off-line”, sometimes at run-time
- Still need to understand how to do it by hand
 - Not every code will have an autotuner
 - Need to know if you want to build autotuners

Optimizing in Practice

- Tiling for registers
 - loop unrolling, use of named “register” variables
- Tiling for multiple levels of cache and TLB
- Exploiting fine-grained parallelism in processor
 - superscalar; pipelining
- Complicated compiler interactions (flags)
- Hard to do by hand (but you’ ll try)
- Automatic optimization an active research area
 - ASPIRE: aspire.eecs.berkeley.edu
 - BeBOP: bebop.cs.berkeley.edu
 - Weekly group meeting Mondays 1pm
 - PHiPAC: www.icsi.berkeley.edu/~bilmes/hipac
in particular tr-98-035.ps.gz
 - ATLAS: www.netlib.org/atlas

Removing False Dependencies

- Using local variables, reorder operations to remove false dependencies

```
a[i] = b[i] + c;  
a[i+1] = b[i+1] * d;
```

false read-after-write hazard
between a[i] and b[i+1]



```
float f1 = b[i];  
float f2 = b[i+1];
```

```
a[i] = f1 + c;  
a[i+1] = f2 * d;
```

With some compilers, you can declare a and b unaliased.

- Done via “restrict pointers,” compiler flag, or pragma

Exploit Multiple Registers

- Reduce demands on memory bandwidth by pre-loading into local variables

```
while( ... ) {  
    *res++ = filter[0]*signal[0]  
            + filter[1]*signal[1]  
            + filter[2]*signal[2];  
    signal++;  
}
```



```
float f0 = filter[0];  
float f1 = filter[1];  
float f2 = filter[2];  
while( ... ) {  
    *res++ = f0*signal[0]  
            + f1*signal[1]  
            + f2*signal[2];  
    signal++;  
}
```

also: register float f0 = ...;

Example is a convolution

Search Over Block Sizes

- Performance models are useful for high level algorithms
 - Helps in developing a blocked algorithm
 - Models have not proven very useful for block size selection
 - too complicated to be useful
 - See work by Sid Chatterjee for detailed model
 - too simple to be accurate
 - Multiple multidimensional arrays, virtual memory, etc.
 - Speed depends on matrix dimensions, details of code, compiler, processor

restrict

- restrict is a c99 keyword
- the pointer is the only thing that accesses the underlying object. It eliminates the potential for pointer aliasing, enabling better optimization by the compiler.
- `void updatePtrs(size_t *restrict ptrA, size_t *restrict ptrB, size_t *restrict val);`
- <https://en.wikipedia.org/wiki/Restrict>

Loop Unrolling

- Expose instruction-level parallelism

```
float f0 = filter[0], f1 = filter[1], f2 = filter[2];
float s0 = signal[0], s1 = signal[1], s2 = signal[2];
*res++ = f0*s0 + f1*s1 + f2*s2;
do {
    signal += 3;
    s0 = signal[0];
    res[0] = f0*s1 + f1*s2 + f2*s0;

    s1 = signal[1];
    res[1] = f0*s2 + f1*s0 + f2*s1;

    s2 = signal[2];
    res[2] = f0*s0 + f1*s1 + f2*s2;

    res += 3;
} while( ... );
```

Expose Independent Operations

- Hide instruction latency
 - Use local variables to expose independent operations that can execute in parallel or in a pipelined fashion
 - Balance the instruction mix (what functional units are available?)

```
f1 = f5 * f9;  
f2 = f6 + f10;  
f3 = f7 * f11;  
f4 = f8 + f12;
```

Locality in Other Algorithms

- The performance of any algorithm is limited by q
 - $q = \text{“computational intensity”} = \text{\#arithmetic_ops} / \text{\#words_moved}$
- In matrix multiply, we increase q by changing computation order
 - increased temporal locality
- For other algorithms and data structures, even hand-transformations are still an open problem
 - Lots of open problems, class projects

Summary of Lecture

- Details of machine are important for performance
 - Processor and memory system (not just parallelism)
 - Before you parallelize, make sure you're getting good serial performance
 - What to expect? Use understanding of hardware limits
- There is parallelism hidden within processors
 - Pipelining, SIMD, etc
- Machines have memory hierarchies
 - 100s of cycles to read from DRAM (main memory)
 - Caches are fast (small) memory that optimize average case
- Locality is at least as important as computation
 - Temporal: re-use of data recently used
 - Spatial: using data nearby to recently used data
- Can rearrange code/data to improve locality
 - Goal: minimize communication = data movement

Some reading for today (see website)

- Sourcebook Chapter 3, (note that chapters 2 and 3 cover the material of lecture 2 and lecture 3, but not in the same order).
- "[Performance Optimization of Numerically Intensive Codes](#)", by Stefan Goedecker and Adolfo Hoisie, SIAM 2001.
- Web pages for reference:
 - [BeBOP Homepage](#)
 - [ATLAS Homepage](#)
 - [BLAS](#) (Basic Linear Algebra Subroutines), Reference for (unoptimized) implementations of the BLAS, with documentation.
 - [LAPACK](#) (Linear Algebra PACKage), a standard linear algebra library optimized to use the BLAS effectively on uniprocessors and shared memory machines (software, documentation and reports)
 - [ScaLAPACK](#) (Scalable LAPACK), a parallel version of LAPACK for distributed memory machines (software, documentation and reports)
- Tuning Strassen's Matrix Multiplication for Memory Efficiency
Mithuna S. Thottethodi, Siddhartha Chatterjee, and Alvin R. Lebeck
in Proceedings of Supercomputing '98, November 1998 [postscript](#)
- "Recursive Array Layouts and Fast Parallel Matrix Multiplication" by Chatterjee et al. IEEE TPDS November 2002.
- Many related papers at bebop.cs.berkeley.edu

Trabalho prático: Multiplicação de Mat

- Ver descrição no Moodle
- Ssequencial
- Trabalhos no máximo em duplas
- Foco em desempenho (mas tem que estar correto)
- Descrever no relatório todas as versões testadas

Conclusões

A maioria das aplicações executam a $< 10\%$ do desempenho “pico” do sistema

q – intensidade computacional: f/m : medida da eficiência do algoritmo

Custo médio de operações aritméticas (f) por número de operações na memória lenta (m)

- Queremos $q \geq t_m/t_f$ (*machine balance – medida da eficiência da máquina*) para que o desempenho do programa dependa unicamente de t_f e f

Para isto procuramos diminuir os acessos a memória lenta.

Algoritmo ingenuo de Mult. Matrizes: $q = 2$

- Blocking (tiling) é uma abordagem básica para aumentar q
 - As técnicas são gerais mas os detalhes dependem da arquitetura (o tamanho do bloco)
 - Técnicas similares possíveis em outras estruturas de dados e algoritmos

Loop Unrolling

- Expose instruction-level parallelism

```
float f0 = filter[0], f1 = filter[1], f2 = filter[2];
float s0 = signal[0], s1 = signal[1], s2 = signal[2];
*res++ = f0*s0 + f1*s1 + f2*s2;
do {
    signal += 3;
    s0 = signal[0];
    res[0] = f0*s1 + f1*s2 + f2*s0;

    s1 = signal[1];
    res[1] = f0*s2 + f1*s0 + f2*s1;

    s2 = signal[2];
    res[2] = f0*s0 + f1*s1 + f2*s2;

    res += 3;
} while( ... );
```


Expose Independent Operations

- Hide instruction latency
 - Use local variables to expose independent operations that can execute in parallel or in a pipelined fashion
 - Balance the instruction mix (what functional units are available?)

```
f1 = f5 * f9 ;  
f2 = f6 + f10 ;  
f3 = f7 * f11 ;  
f4 = f8 + f12 ;
```

- the recursive layout works well for any cache size

- The index calculations to find $A[i,j]$ are expensive

Strassen's Matrix Multiply

- The traditional algorithm (with or without tiling) has $O(n^3)$ flops
- Strassen discovered an algorithm with asymptotically lower flops
 - $O(n^{2.81})$
- Consider a 2x2 matrix multiply, normally takes 8 multiplies, 7 adds
 - Strassen does it with 7 multiplies and 18 adds

$$\text{Let } M = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$\text{Let } p_1 = (a_{12} - a_{22}) * (b_{21} + b_{22})$$

$$p_5 = a_{11} * (b_{12} - b_{22})$$

$$p_2 = (a_{11} + a_{22}) * (b_{11} + b_{22})$$

$$p_6 = a_{22} * (b_{21} - b_{11})$$

$$p_3 = (a_{11} - a_{21}) * (b_{11} + b_{12})$$

$$p_7 = (a_{21} + a_{22}) * b_{11}$$

$$p_4 = (a_{11} + a_{12}) * b_{22}$$

$$\text{Then } m_{11} = p_1 + p_2 - p_4 + p_6$$

$$m_{12} = p_4 + p_5$$

$$m_{21} = p_6 + p_7$$

$$m_{22} = p_2 - p_3 + p_5 - p_7$$

Extends to nxn by divide&conquer

Strassen (continued)

$$\begin{aligned} T(n) &= \text{Cost of multiplying } n \times n \text{ matrices} \\ &= 7 * T(n/2) + 18 * (n/2)^2 \\ &= O(n^{\log_2 7}) \\ &= O(n^{2.81}) \end{aligned}$$

- Asymptotically faster
 - Several times faster for large n in practice
 - Cross-over depends on machine
 - Available in several libraries
- Caveats
 - Needs more memory than standard algorithm
 - Can be less accurate because of roundoff error
 - Current world's record is $O(n^{2.376..})$

Locality in Other Algorithms

- The performance of any algorithm is limited by q
- In matrix multiply, we increase q by changing computation order
 - **increased temporal locality**

Minimize Pointer Updates

- Replace pointer updates for strided memory addressing with constant array offsets

```
f0 = *r8; r8 += 4;  
f1 = *r8; r8 += 4;  
f2 = *r8; r8 += 4;
```



```
f0 = r8[0];  
f1 = r8[4];  
f2 = r8[8];  
r8 += 12;
```

Pointer vs. array expression costs may differ.

- Some compilers do a better job at analyzing one than the other

OpenMP: boas práticas

1. Try to change number of threads in parallel region after start of region: The number of threads carrying out a parallel region can only be changed before the start of the region. It is therefore a mistake to attempt to change this number from inside the region.
2. Attempt to change loop variable while in `#pragma omp for`: It is explicitly forbidden in the specification to change the loop variable from inside the loop.
3. Use of critical when atomic would be sufficient: There are special cases when synchronisation can be achieved with a simple atomic construct. Not using it in this case leads to potentially slower programs and is therefore a performance mistake.
4. Use of unnecessary critical: The mistake here is to protect memory accesses with a critical construct, although they need no protection (e.g. on private variables or on other occasions, where only one thread is guaranteed to access the location).
5. Every read to a shared variable must be preceded by a flush, except in very rare edge-cases
6. The most frequently made and most severe mistake : to not avoid concurrent access to the same memory location. A way to make novice programmers aware of the issue is to use the available tools to diagnose OpenMP programs. For example, the Intel Thread Checker .

Common Mistakes in OpenMP and How To Avoid Them, A Collection of Best Practices , Michael Suß and Claudia Leopold

Questions You Should Be Able to Answer

1. What is the key to understand algorithm efficiency in our simple memory model?
2. What is the key to understand machine efficiency in our simple memory model?
3. What is tiling?
4. Why does block matrix multiply reduce the number of memory references?
5. What are the BLAS?
6. Why does loop unrolling improve uniprocessor performance?

Summary

- Performance programming on uniprocessors requires
 - understanding of memory system
 - understanding of fine-grained parallelism in processor
- Simple performance models can aid in understanding
 - Two ratios are key to efficiency (relative to peak)
 - 1.computational intensity of the algorithm:
 - $q = f/m = \# \text{ floating point operations} / \# \text{ slow memory references}$
 - 2.machine balance in the memory system:
 - $t_m/t_f = \text{time for slow memory reference} / \text{time for floating point operation}$
- Want $q > t_m/t_f$ to get half machine peak
- Blocking (tiling) is a basic approach to increase q
 - Techniques apply generally, but the details (e.g., block size) are architecture dependent
 - Similar techniques are possible on other data structures and algorithms