CS267 Lecture 2

Single Processor Machines: Memory Hierarchies and Processor Features

Case Study: Tuning Matrix Multiply

Kathy Yelick

https://sites.google.com/lbl.gov/cs267-spr2018/

Motivation

- Parallel computing is about performance
  - Distributed computing is a different thing
- Most applications run at < 10% of the "peak" performance
  - Peak is the maximum the hardware can physically execute
- Much of the performance is lost on a single processor
  - The code running on one processor often runs at only 10-20% of the processor peak
- Most of that loss is in the memory system
  - Moving data takes much longer than arithmetic and logic
- 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

Possible conclusions to draw from today's lecture

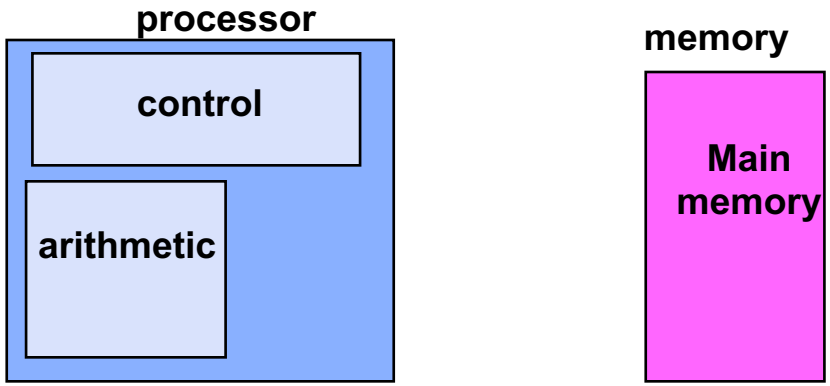
- "Computer architectures are fascinating—I really want to understand why apparently simple programs can behave in such complex ways!"
- "These optimizing algorithms are fascinating—I want to learn more about algorithms for complicated hardware."
- "I hope that most of the time I can call libraries so I don't have to worry about this!"
- "I wish the compiler would handle everything for me."
- "I would like to write a compiler that would handle all of these details."
- "I want to understand how Meltdown/Spectre work"

Outline

- Costs in modern processors
  - Idealized models and actual costs
- Memory hierarchies
- Parallelism in a single processor
- Case study: Matrix multiplication

Idealized Uniprocessor Model

- Processor names variables:
  - Integers, floats, pointers, arrays, structures, etc.
- Processor performs operations on those variables:
  - Arithmetic, logical operations, etc.
- Processor controls the order, as specified by program
  - Branches (if), loops, function calls, etc.

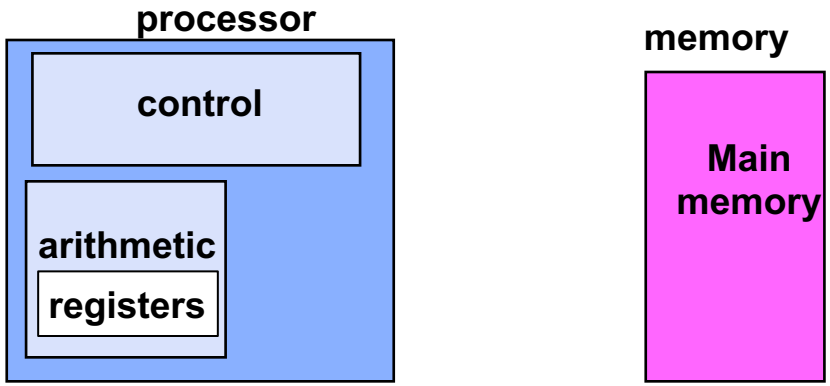


- *Idealized Cost*
  - Each operation has roughly the same cost

    add, multiply, etc.

Slightly Less Idealized Uniprocessor Model

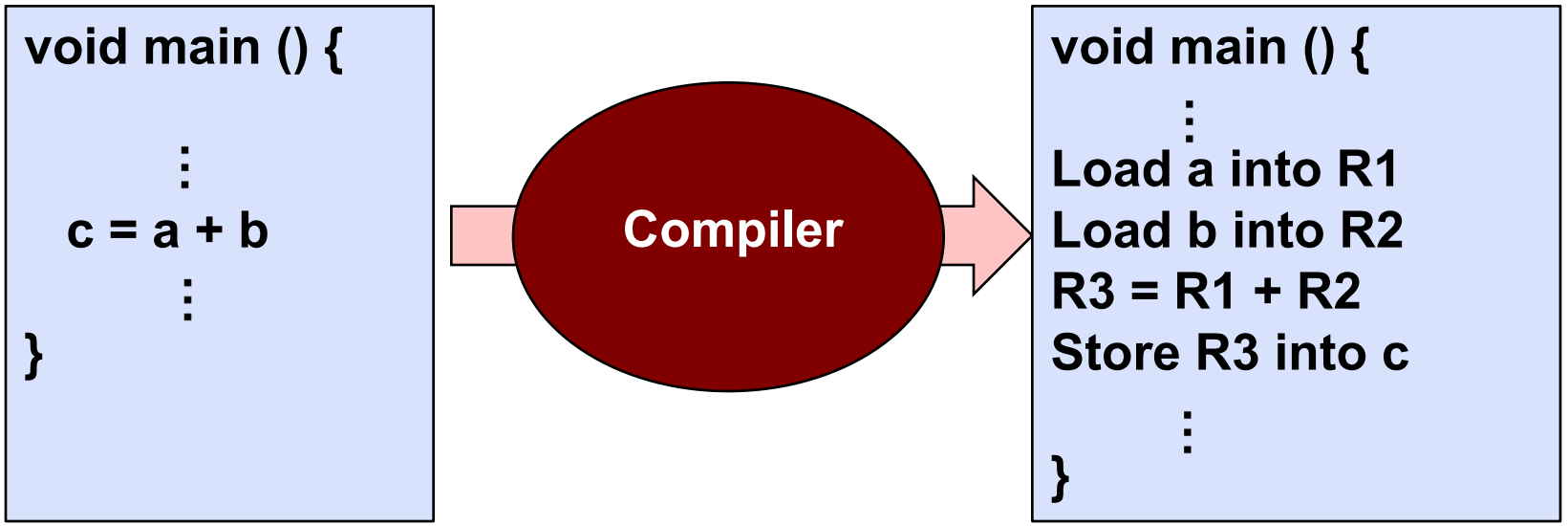
- Processor names variables:
  - Integers, floats, pointers, arrays, structures, etc.
  - These are really words, e.g., 64-bit doubles, 32-bit ints, bytes, etc.
- Processor performs operations on those variables:
  - Arithmetic, logical operations, etc.
  - Only performs these operations on values in registers
- Processor controls the order, as specified by program
  - Branches (if), loops, function calls, etc.



- *Idealized Cost*
  - Each operation has roughly the same cost

    add, multiply, etc.

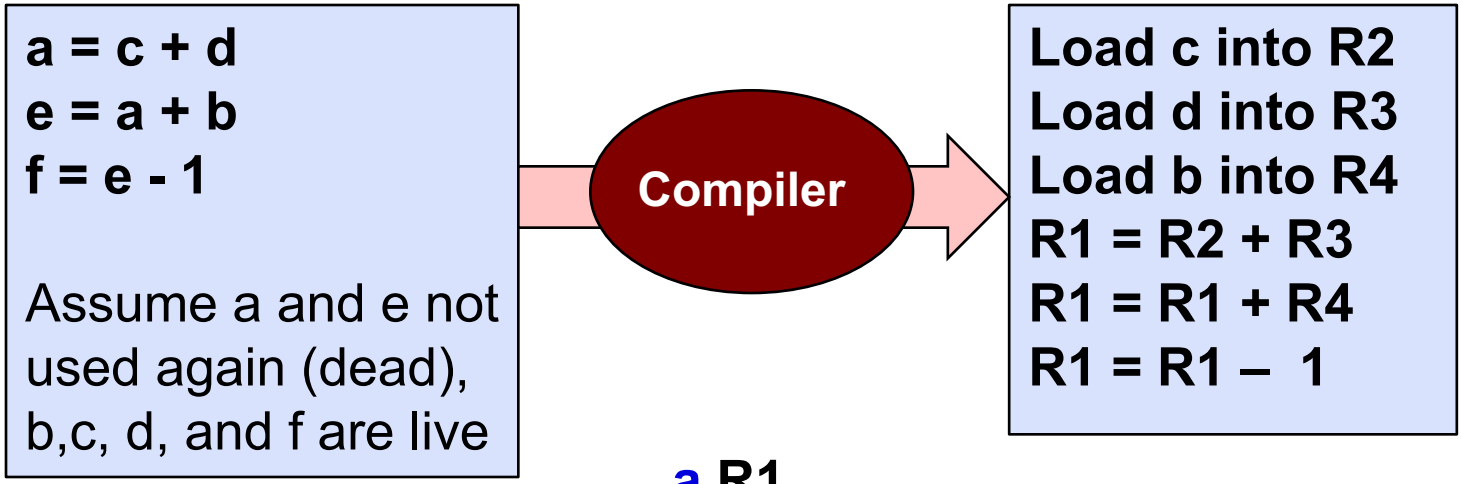
Compilers and assembly code

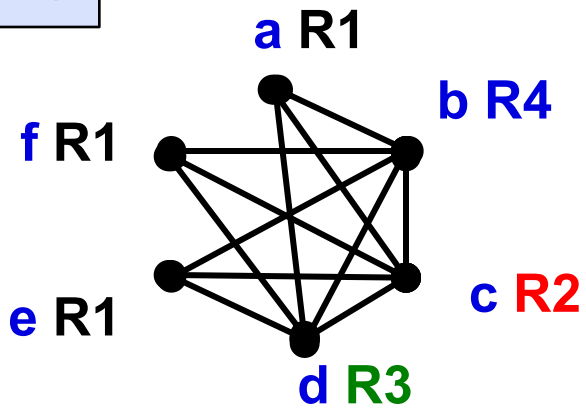


- Compilers for languages like C/C++ and Fortran:
  - Check that the program is legal
  - Translate into assembly code
  - Optimizes the generated code

Compilers Manage Memory and Registers

- **Compiler performs "register allocation" to decide when to load/store and when to reuse**





Register allocation in first Fortran compiler in 1950s, graph coloring in 1980. JITs may use something cheaper

Compiler Optimizes Code

- **Compiler performs a number of optimizations:**
  - **Unrolls loops (because control isn't free)**
  - **Fuses loops (merge two together)**
  - **Interchanges loops (reorder)**
  - **Eliminates dead code (the branch never taken)**
  - **Reorders instructions to improve register reuse and more**
  - **Strength reduction (turns expensive instruction, e.g., multiply by 2, into cheaper one, shift left)**

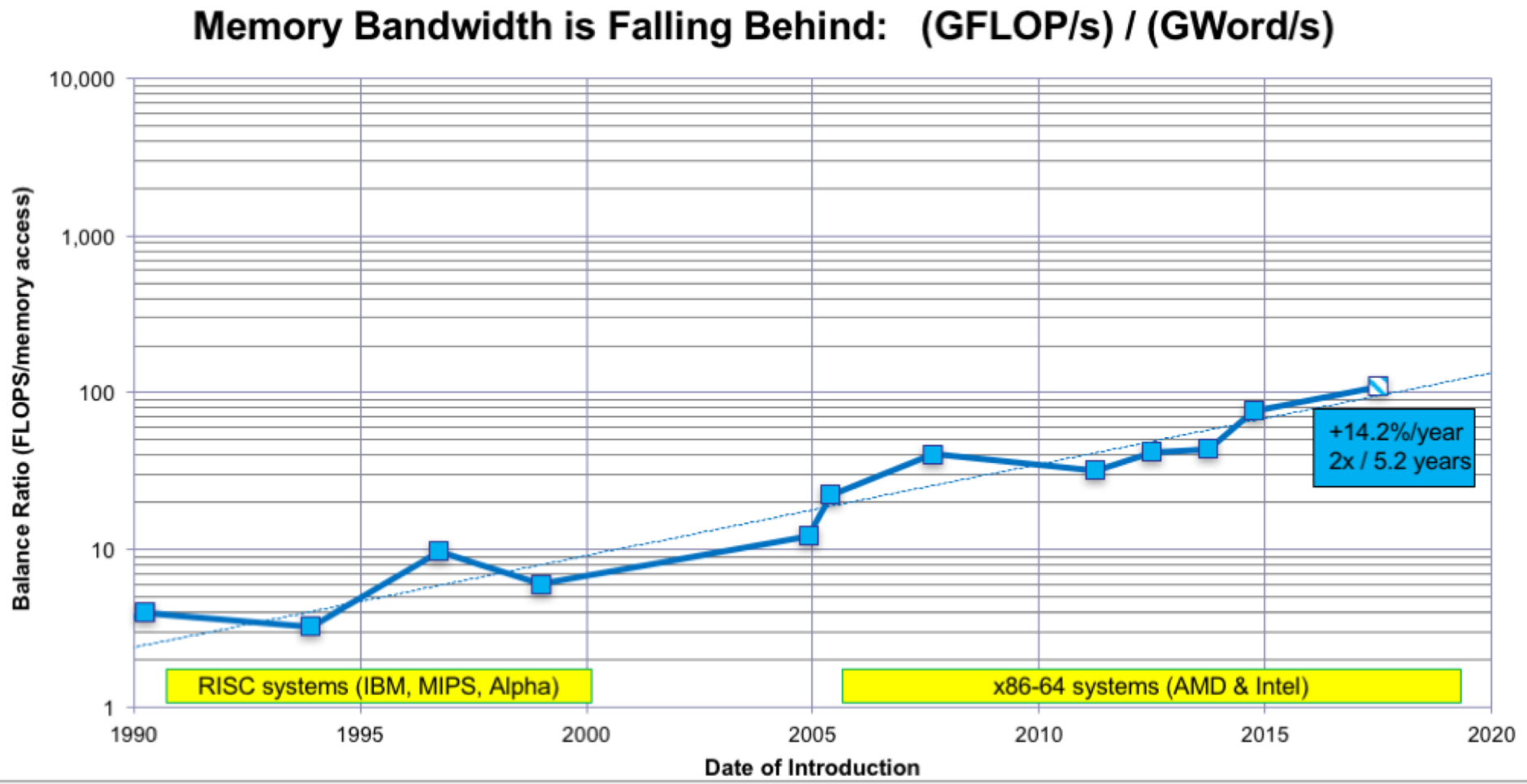
- **The compiler manages operations (load/store) that are important to performance but we don't "see" in code.**

- **Why is this your problem?**
  - **Because sometimes it does the best thing possible**
  - **But other times it does not…**

Outline

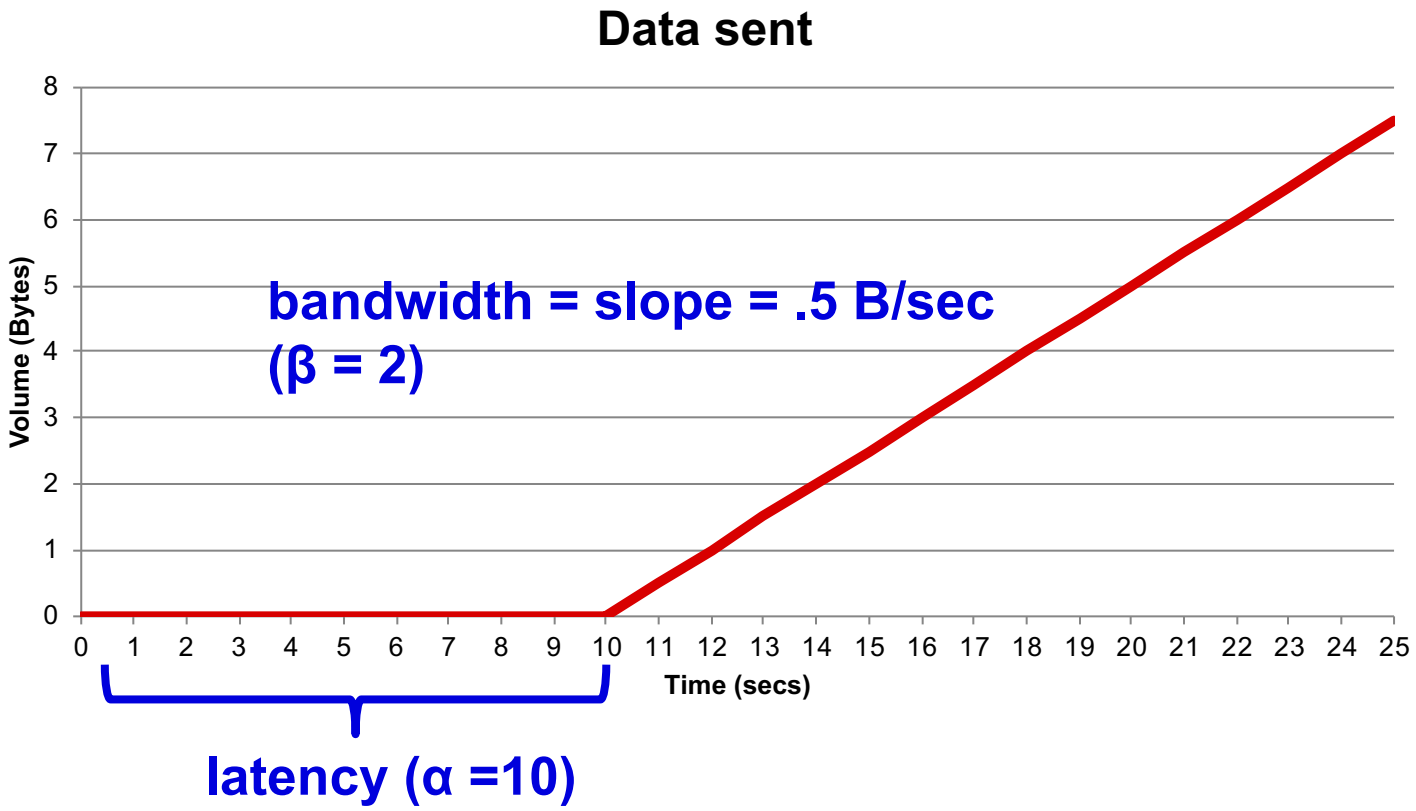
- Costs in modern processors
- Memory hierarchies
  - Temporal and spatial locality
  - Basics of caches
  - Use of microbenchmarks to characterize performance
- Parallelism in a single processor
- Case study: Matrix multiplication

Memory Bandwidth Gap

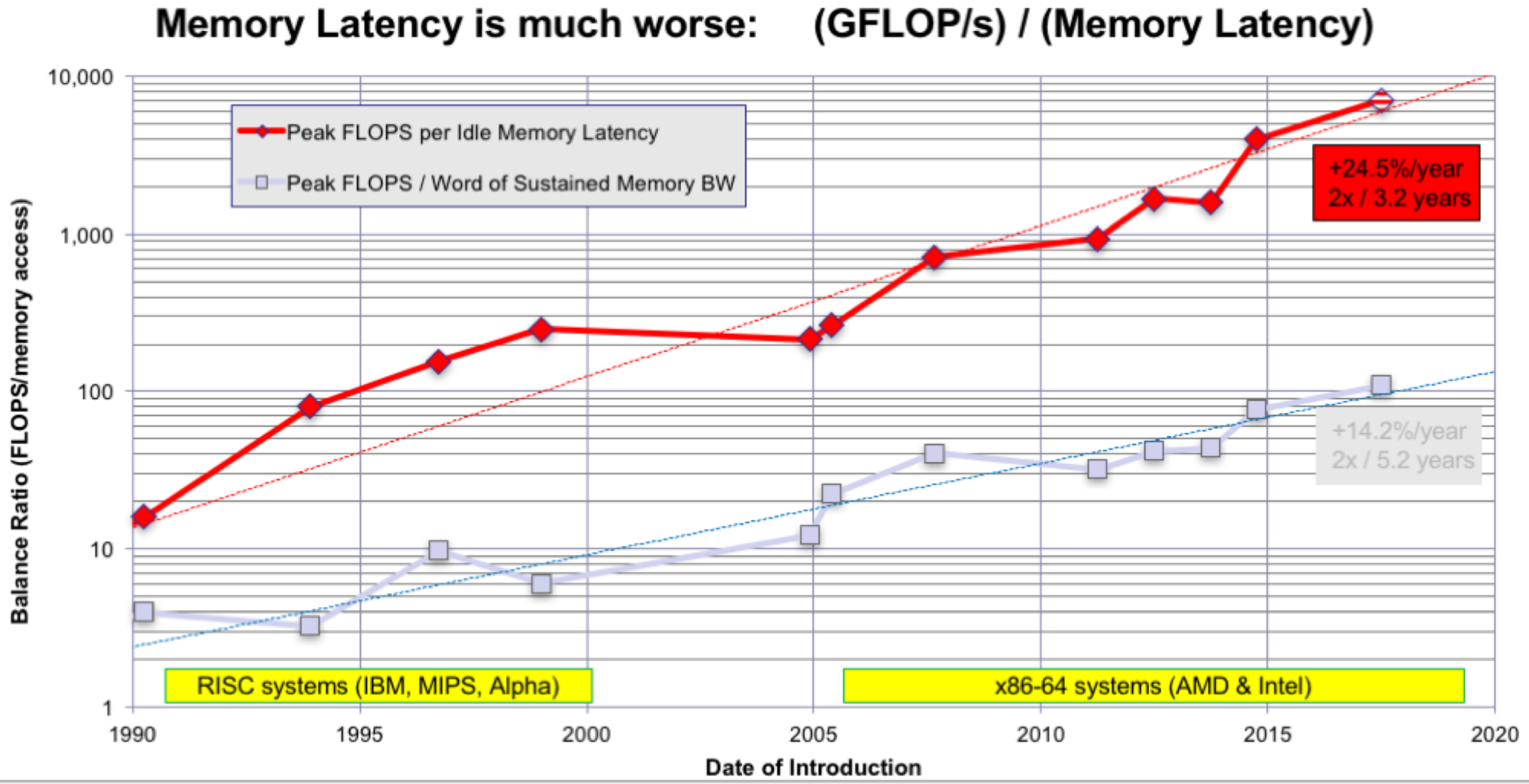


Latency vs Bandwidth

- Previous graph show bandwidth gap
- Many applications are limited not by memory bandwidth, but by latency

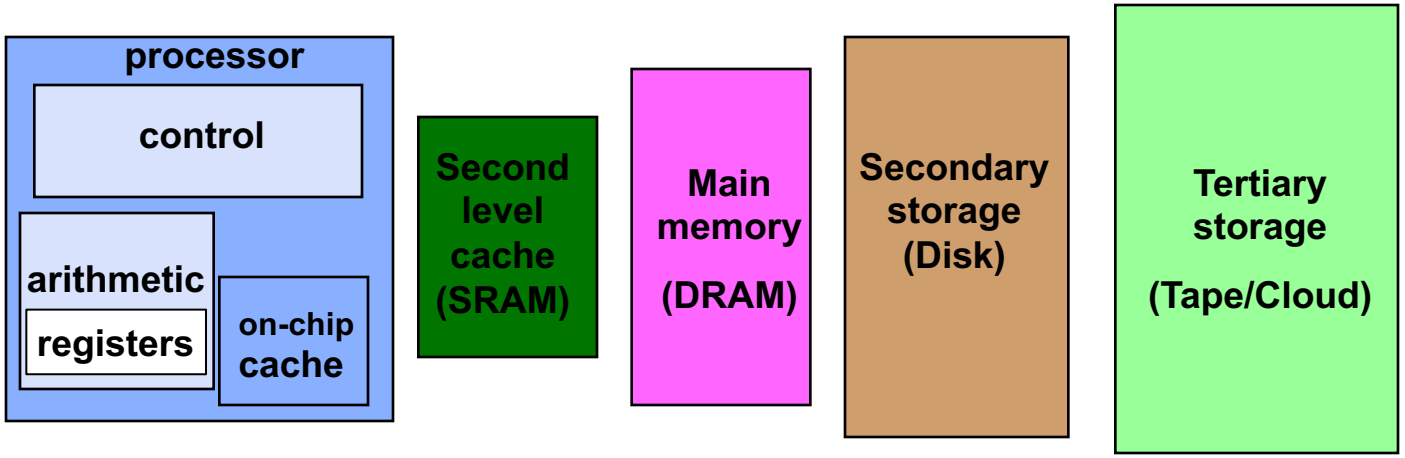


Memory Latency Gap is Worse



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

Cache Basics

- **Cache is fast (expensive) memory which keeps copy of data in main memory; hidden from software**
  - Simplest example: data at memory address xxxxx1101 is stored at cache location 1101
- **Cache hit: in-cache memory access—cheap**
- **Cache miss: non-cached memory access—expensive**
  - Need to access next, slower level of cache
- **Cache line size: # of bytes loaded together in one entry**
  - If either xxxxx1100 or xxxxx1101 is loaded, both are
- **Associativity**
  - **direct-mapped: only 1 address (line) in a given range in cache**
    - Data from any address xxxxx1101 stored at cache location 1101
  - ***n*-way associate: $n \geq 2$ lines with different addresses can be stored**
    - Data from up to n addresses xxxxx1101 can be stored at n spots in cache location 1101

Why Have Multiple Levels of Cache?

- On-chip vs. off-chip
  - **On-chip caches are faster, but limited in size**
- A large cache has delays
  - **Hardware to check longer addresses in cache takes more time**
  - **Associativity, which gives a more general set of data in cache, also takes more time**

- Some examples:
  - **Cray T3E eliminated one cache to speed up misses**
  - **Intel Haswell (Cori Ph1) uses a Level 4 cache as "victim cache"**
- There are other levels of the memory hierarchy
  - **Register, pages (TLB, virtual memory), …**
  - **And it isn't always a hierarchy**

Approaches to Handling Memory Latency

- Eliminate memory operations by saving values in small, fast memory (cache or registers) and reusing them (bandwidth filtering)
  - **need temporal locality in program**
- Take advantage of better bandwidth by getting a chunk of memory into cache (or vector registers) and using whole chunk
  - **need spatial locality in program**

concurrency

- Take advantage of better bandwidth by allowing processor to issue multiple reads or writes with a single instruction
  - **vector operations require access set of locations (typically neighboring), requires they are independent**
- Take advantage of better bandwidth by allowing processor to issue reads/writes in parallel with other reads/writes/operations
  - **prefetching issues read hint**
  - **delayed writes (write buffering) stages writes for later operation**
  - **both require that nothing dependent is happening (parallelism)**

How much concurrency do you need?

- To run at bandwidth speeds rather than latency
- Little's Law from queuing theory says:

  concurrency = latency * bandwidth

- In my earlier example with

  latency = 10 sec

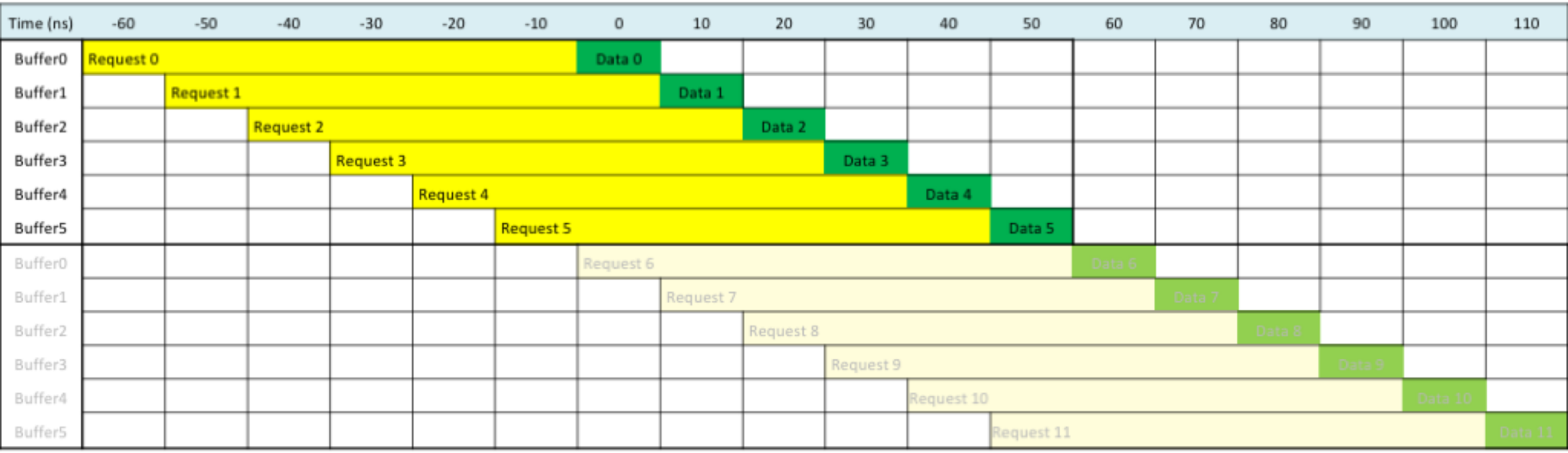
  bandwidth = 2 Bytes/sec

- Requires 20 bytes in flight concurrently to reach bandwidth speeds
- That means finding 20 independent things to issue

More realistic example

Little's Law: illustration for 2005-era Opteron processor

60 ns latency, 6.4 GB/s (=10ns per 64B cache line)



- 60 ns * 6.4 GB/s = 384 Bytes = 6 cache lines
- To keep the pipeline full, there must always be 6 cache lines "in flight"
- Each request must be launched at least 60 ns before the data is needed

Stream benchmark for measuring bandwidth

- Stream benchmark (also due to McCalpin)
- Four kernels, all for i = 1 to N:
  - `Copy:  C[i] = A[i];`
  - `Scale:  B[i]  = scalar * C[i];`
  - `Add:  C[i] = A[i] + B[i];`
  - `Triad:  A[i] = B[i] + scalar * C[i];`
- N chose so that array is much larger than cache
- Repeated ~10 times (ignoring first) to “warm cache
- Min/Av/Max timing report

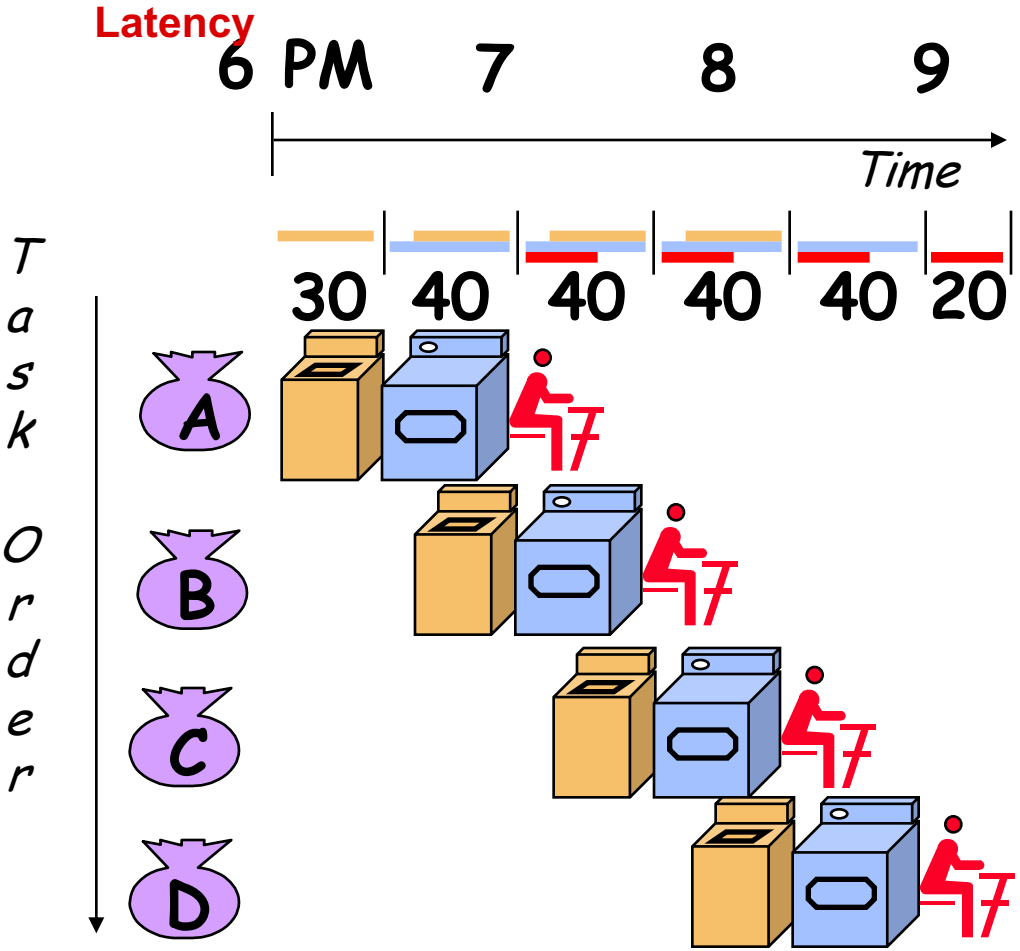
Outline

- Idealized and actual costs in modern processors
- Memory hierarchies
- Parallelism within single processors
  - Instruction Level Parallelism (ILP)
  - SIMD units
  - Special Instructions (FMA)
- Case study: Matrix Multiplication

What is Pipelining?

Dave Patterson's Laundry example: 4 people doing laundry

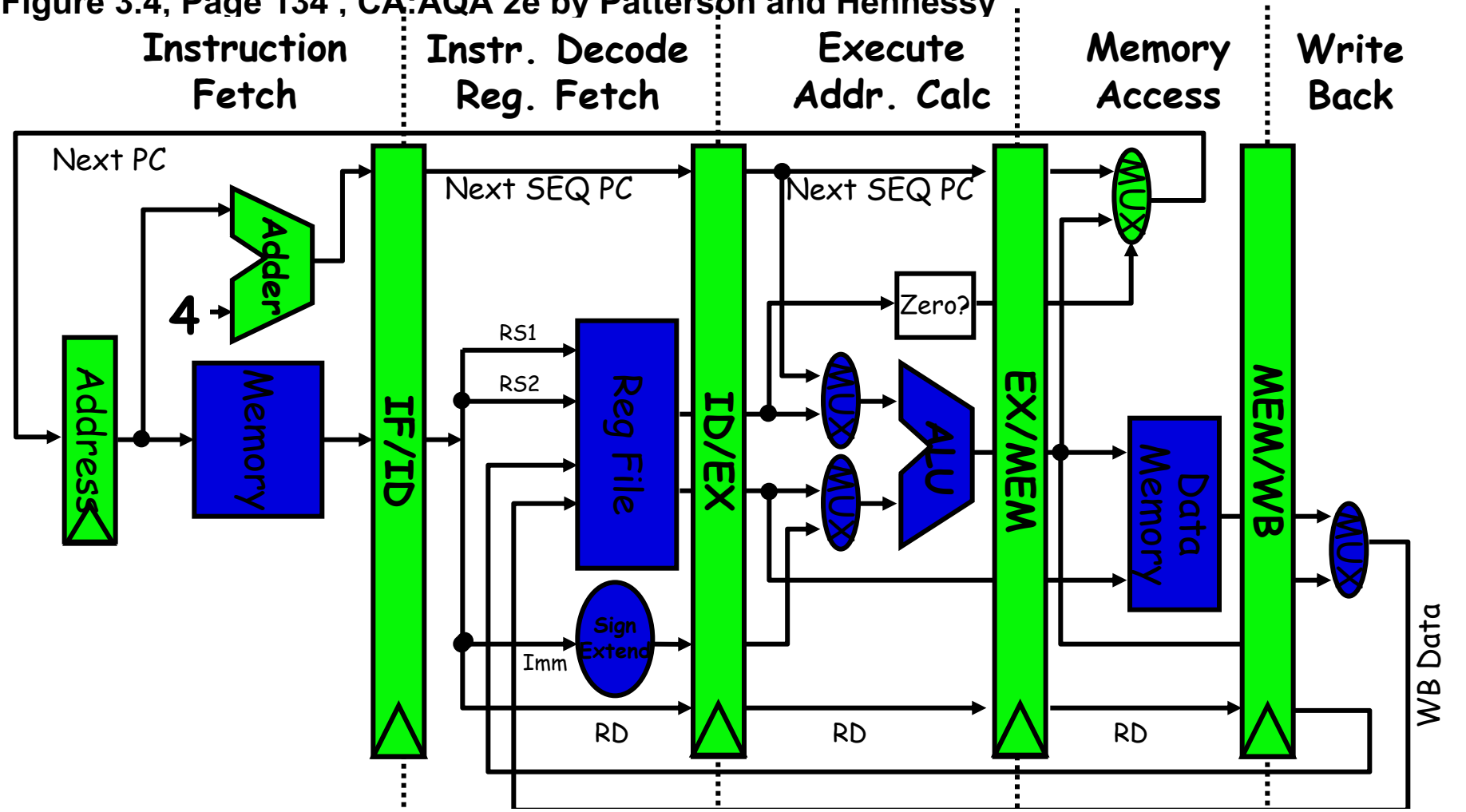
wash (30 min) + dry (40 min) + fold (20 min) = 90 min



- In this example:
  - Sequential execution takes 4 * 90min = 6 hours
  - Pipelined execution takes 30+4*40+20 = 3.5 hours
- Bandwidth = loads/hour
- BW = 4/6 l/h w/o pipelining
- BW = 4/3.5 l/h w pipelining
- BW <= 1.5 l/h w pipelining, more total loads
- Pipelining doesn't change latency (90 min)
- Bandwidth limited by slowest pipeline stage
- Speedup <= # of stages

Example: 5 Steps of MIPS Datapath

Figure 3.4, Page 134 , CA:AQA 2e by Patterson and Hennessy



- Pipelining is also used within arithmetic units
  - a fp multiply may have latency 10 cycles, but throughput of 1/cycle

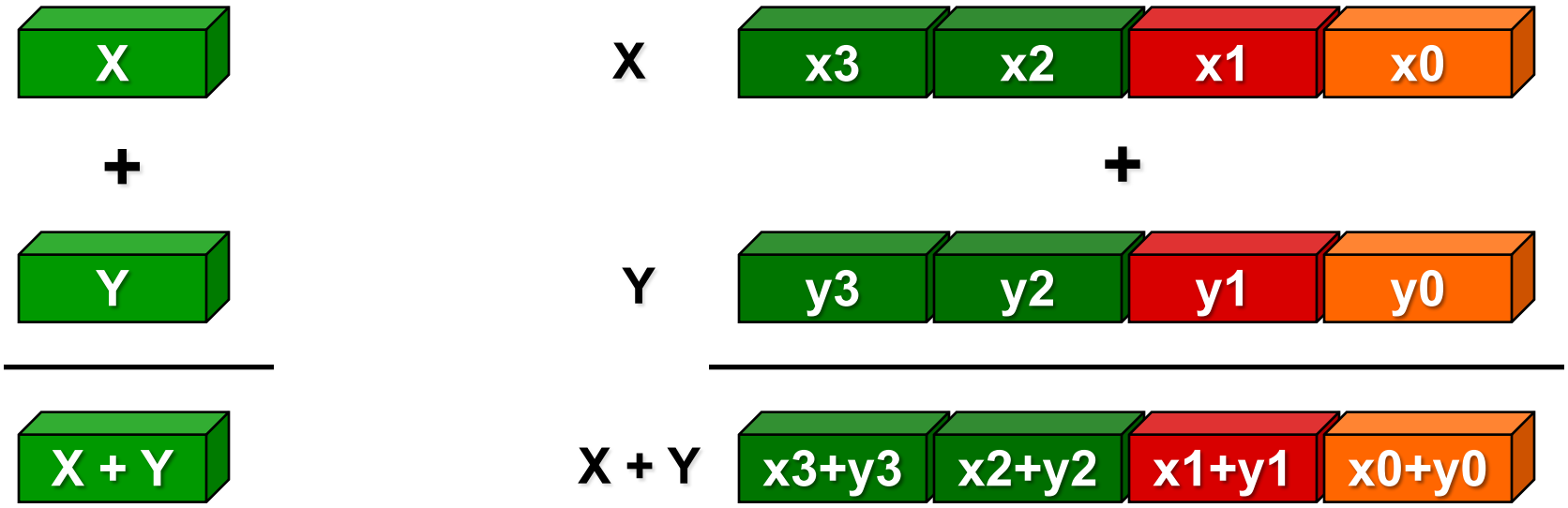
SIMD: Single Instruction, Multiple Data

Scalar processing

- traditional mode
- one operation produces one result

SIMD processing: vectors

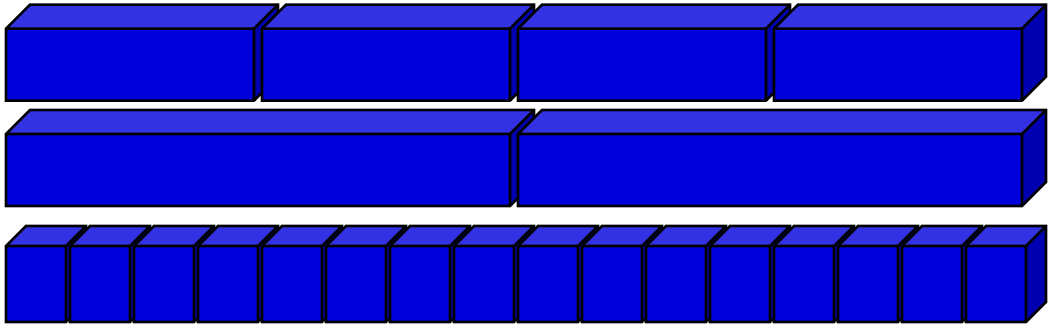
- Sandy Bridge: AVX (256 bit)
- Haswell: AVX2 (256 bit w/ FMA)
- KNL: AVX-512 (512 bit)



http://www.nersc.gov/users/computational-systems/edison/programming/vectorization/

SSE / SSE2 SIMD on Intel

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



4x floats

2x doubles

16x bytes

- 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
  - Some instructions to move data around from one part of register to another
- Similar on GPUs, vector processors (but many more simultaneous operations)

Data Dependencies Limit Parallelism

- Parallelism can get the wrong answer if instructions execute out of order

Types of dependencies

- RAW: Read-After-Write
  - X = A ; B = X;
- WAR: Write-After-Read
  - A = X ; X = B;
- WAW: Write-After-Write
  - X = A ; X = B;

- No problem / dependence for RAR: Read-After-Read

Special Instructions

- Multiply followed by add is very common on programs

  x = y + c * z

- Useful in matrix multiplication
- Fused Multiply-Add (FMA) instructions:
  - Performs multiply/add, at the same rate as + or * alone
  - And does so with a single rounding step

  x = round(c * z+ y)

What does this mean to you?

- In theory, the compiler understands all of this
  - It will rearrange instructions to maximizes parallelism, uses FMAs and SIMD
  - While preserving dependencies

- 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 ☹

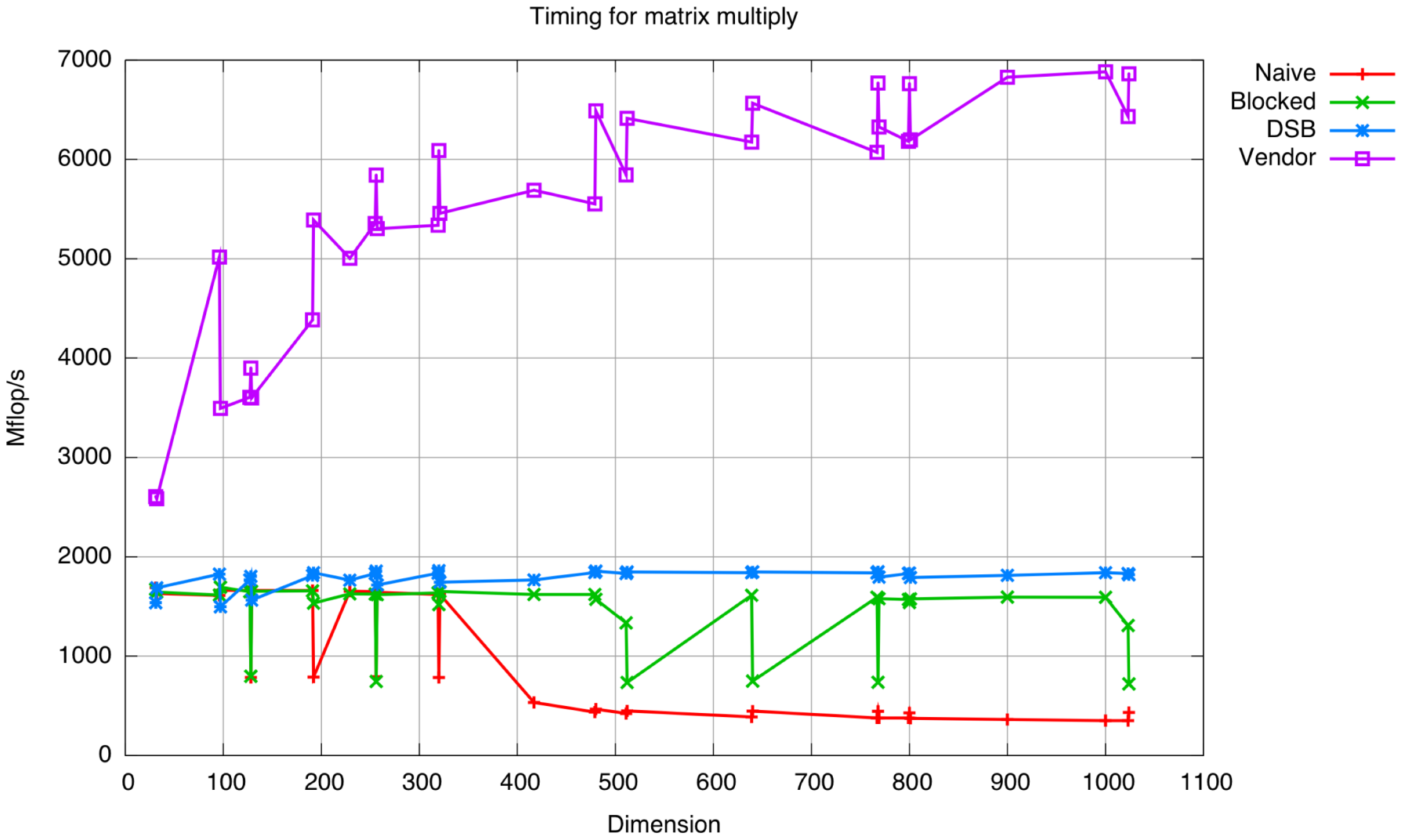
Outline

- Idealized and actual costs in modern processors
- Memory hierarchies
- 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 (*continued next time*)

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 motifs of parallel computing
  - Closely related to other algorithms, e.g., transitive closure on a graph using Floyd-Warshall
  - **And dominates training time in deep learning (CNNs)**
- Optimization ideas can be used in other problems
- The best case for optimization payoffs
- The most-studied algorithm in high performance computing

Performance from Optimization



Graph from David Bindle (DSB)

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)

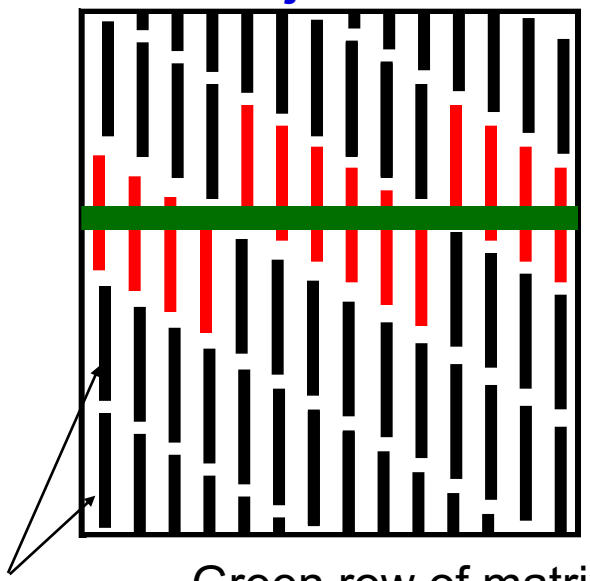
Column major

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



cachelines

Green row of matrix is stored in red cachelines

- Column major (for now)

Figure source: Larry Carter, UCSD

Using a Simple Model of Memory to Optimize

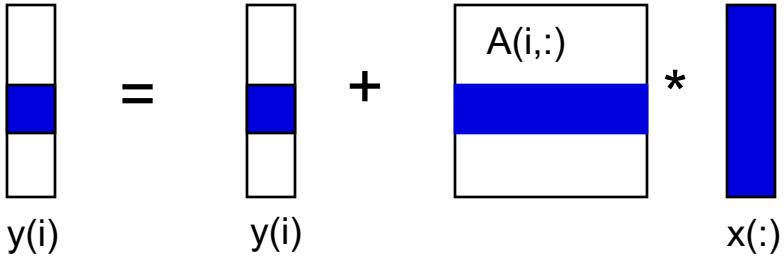
- Assume just 2 levels in the hierarchy, fast and slow
- All data initially in slow memory
  - m = number of memory elements (words) moved between fast and slow memory
  - t_m = time per slow memory operation
  - f = number of arithmetic operations
  - t_f = time per arithmetic operation << t_m
  - $q = f / m$ average number of flops per slow memory access
- Minimum possible time = $f * t_f$ when all data in fast memory
- Actual time
  - $f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/q)$
- 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 = $f / m \approx 2$

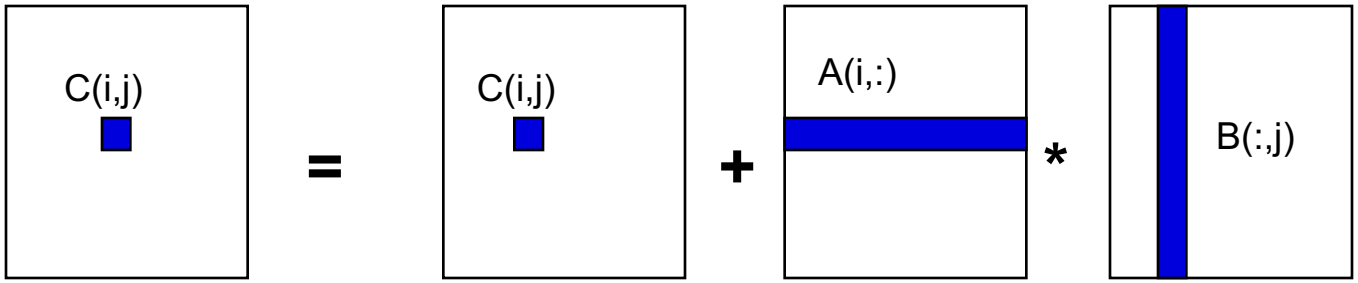
- Matrix-vector multiplication limited by slow memory speed

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)
```

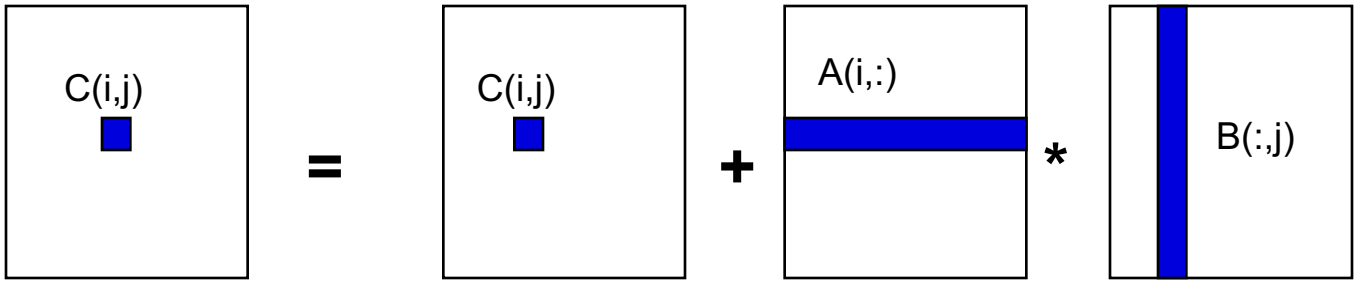
Algorithm has $2*n^3 = O(n^3)$ Flops and operates on $3*n^2$ words of memory

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

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

$+ n^2$ to read each row of A once

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

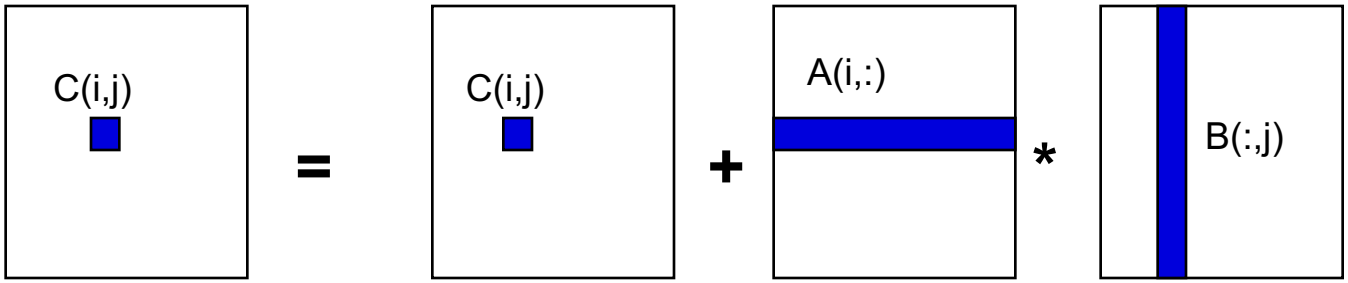
$= n^3 + 3n^2$

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

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

Inner two loops are just matrix-vector multiply, of row i of A times B

Similar for any other order of 3 loops



Blocked (Tiled) Matrix Multiply

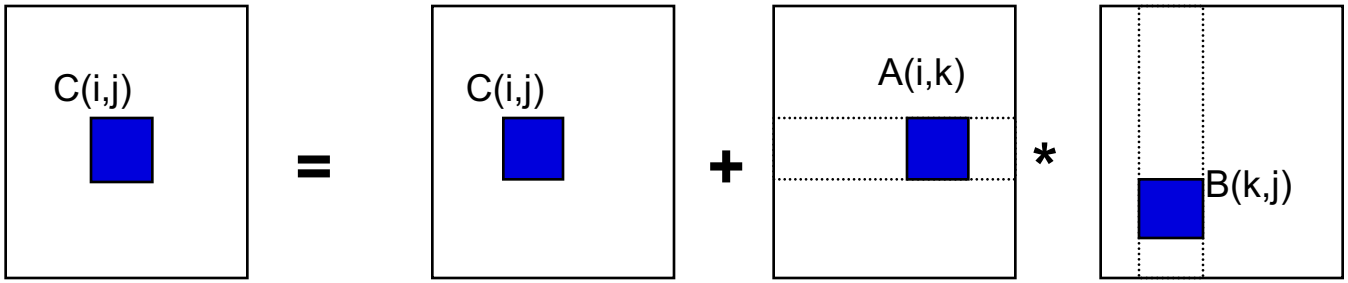
Consider A,B,C to be N-by-N matrices of b-by-b subblocks where
b=n / N is called the block size

```
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}
```

cache does this automatically

3 nested loops inside

block size = loop bounds

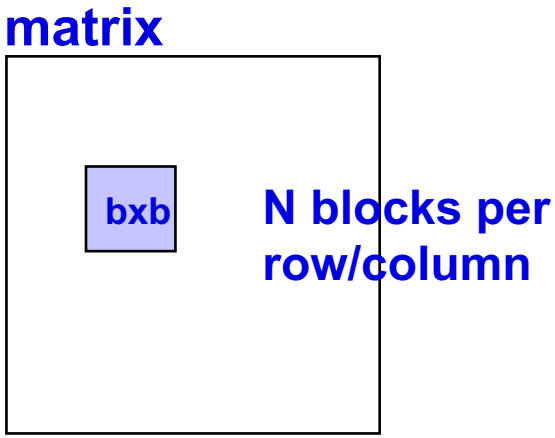


Tiling for registers (managed by you/compiler) or caches (hardware)

Computational Intensity (q) of Tiled Matrix Multiply

Recall:

- m is memory traffic
- f is # floating point operations ($2n^3$)
- q = f / m is computational intensity



So:

m = $N*n^2$ read each block of B N^3 times ($N^3 * b^2 = N^3 * (n/N)^2 = N*n^2$)

+ $N*n^2$ read each block of A N^3 times

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

= $(2N + 2) * n^2$

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

$\approx n / N = b$

Make b as large as possible (up to ~square root of cache size)

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

Computational Intensity – Flops/Cycle

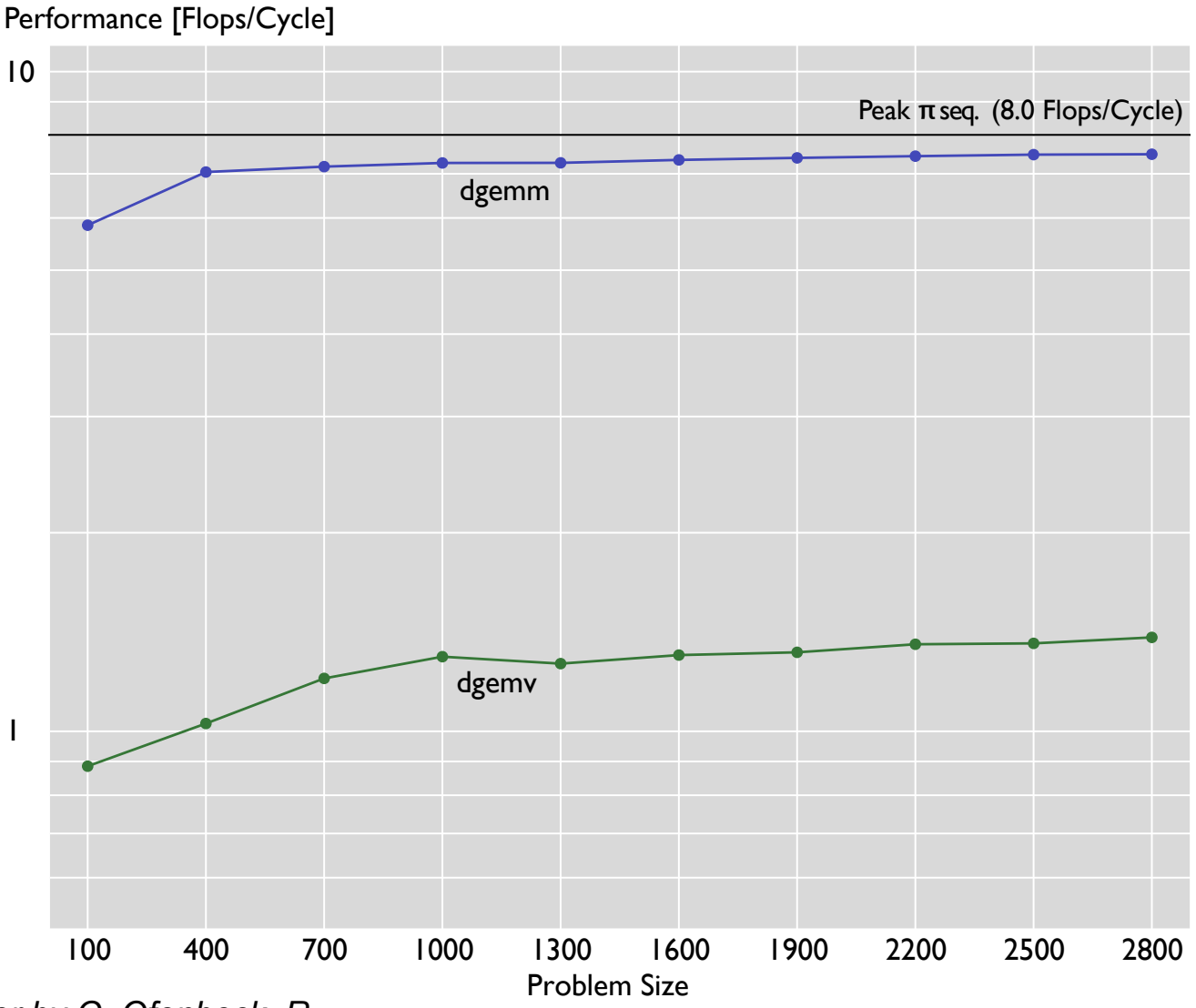


Image and paper by G. Ofenbeck, R. Steinman, V. Caparrós Cabezas, D. Spampinato, M. Püschel

Theory: Communication lower bounds for Matmul

Theorem (Hong & Kung, 1981):

Any reorganization of matmul (using only associativity) has computational intensity $q = O((M_{fast})^{1/2})$, so

#words moved between fast/slow memory $= \Omega (n^3 / (M_{fast})^{1/2})$

Useful work max:
Matmul does $O(n^3)$ work on $O(n^2)$ data, in cache $n = \sqrt{M}$
so $q = O(\sqrt{M})$

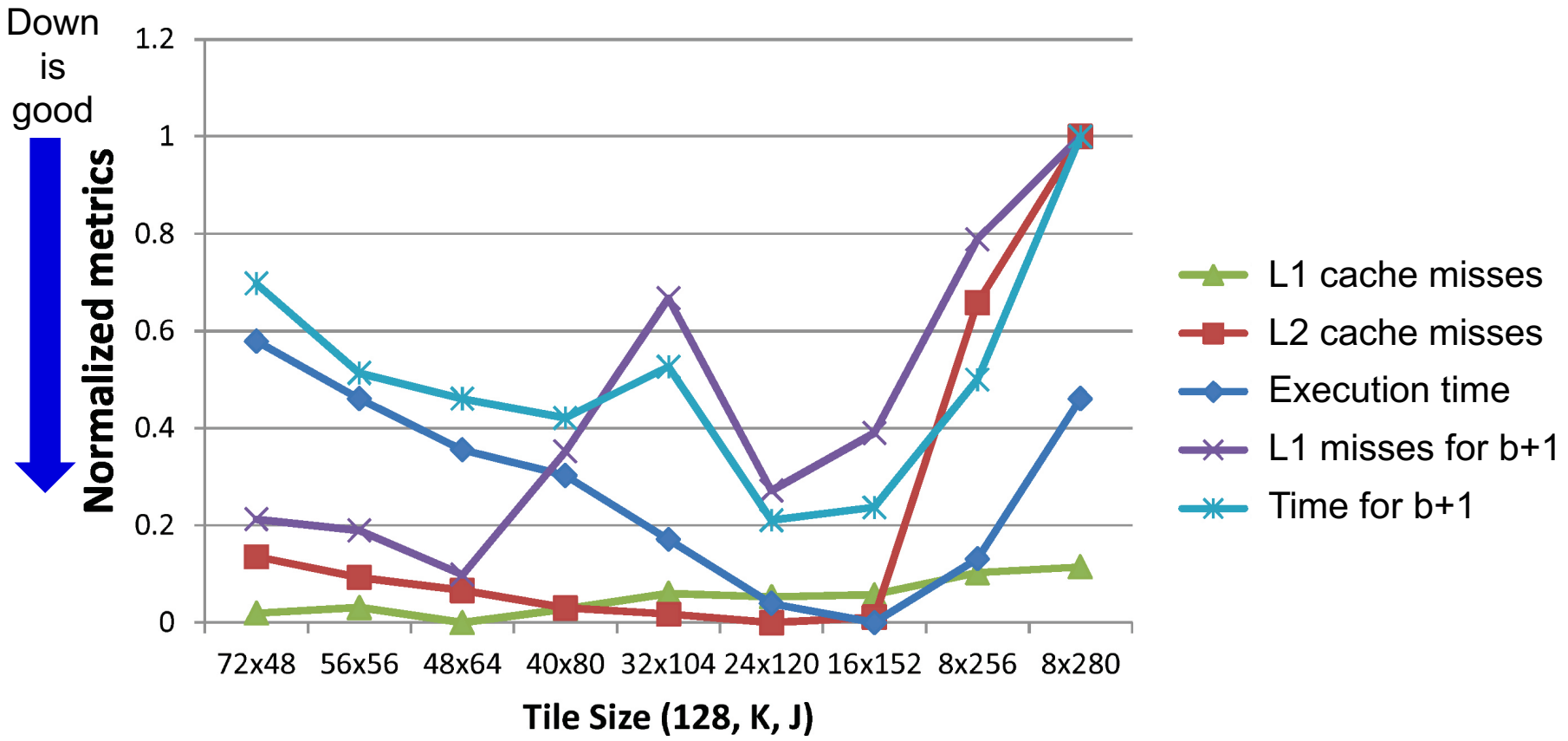
Data transferred min:

$$\frac{\text{All work}}{\text{Useful work in cache}} = \Omega (n^3 / M^{1/2})$$

- Cost also depends on the number of "messages" (e.g., cache lines)
  - #messages $= \Omega (n^3 / M_{fast}^{3/2})$
- Tiled matrix multiply (with tile size $= M_{fast}^{1/2} / 3$) achieves this lower bound
- Lower bounds extend to similar programs nested loops accessing arrays

Practice: Tile Size Selection for Cache-Aware Tiling

- Maximize b, but small enough to fit in cache (or in registers)
  - Avoid interference: depends on n, b, cache associativy, etc.
  - Not necessarily square block (row / column accesses different)

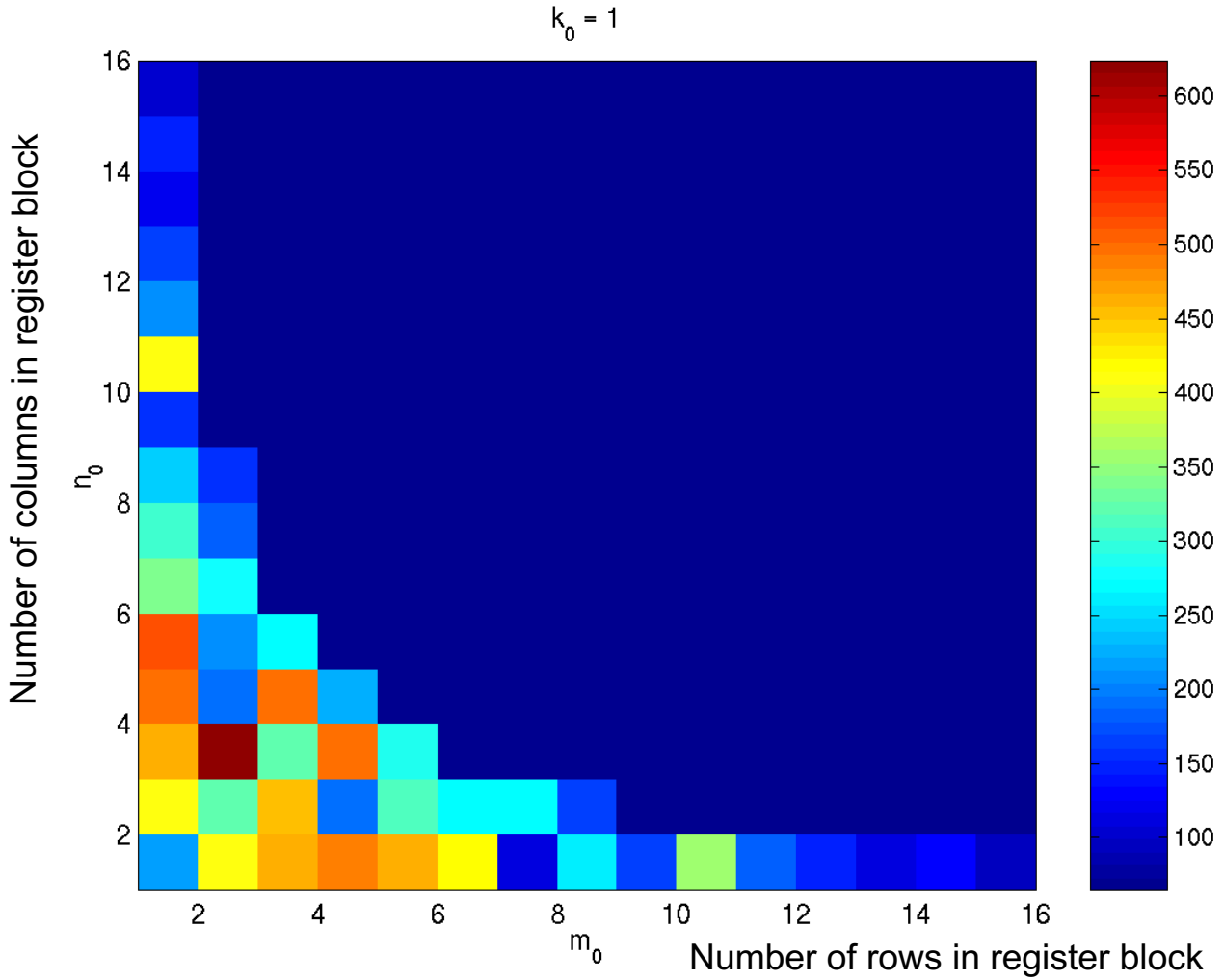


Hundreds of papers on this; above from [Mehtra, Beeraka, Yew, 2013]

Tuning Code in Practice

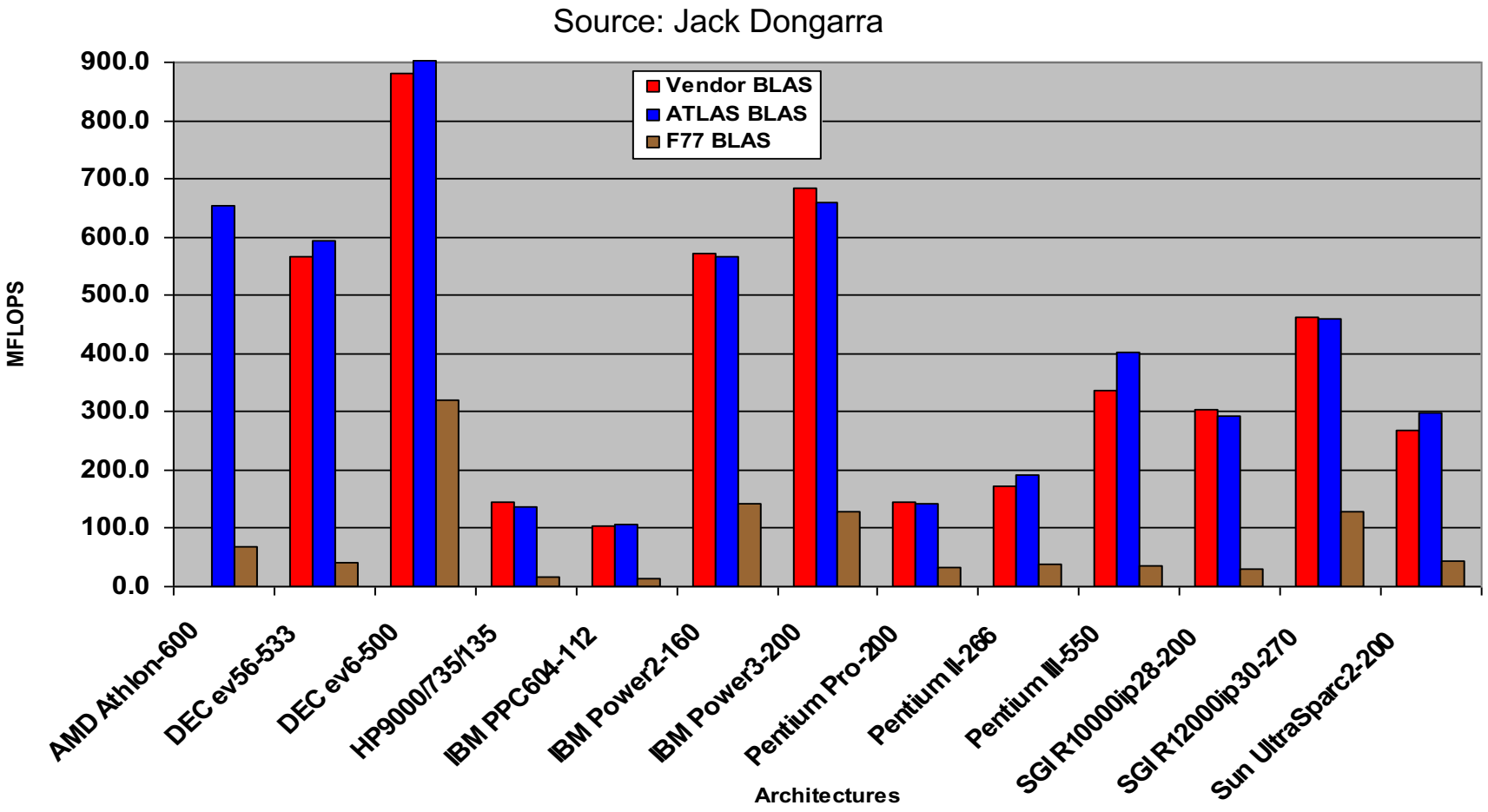
- Tuning code can be tedious
  - Cache has associativity, alignment issues
- Approach #1: Analytical performance models
  - Use model (of cache, memory costs, etc.) to select best code
  - But model needs to be both simple and accurate ☹
- Approach #2: "Autotuning"
  - Let computer generate large set of possible code variations, and search for the fastest ones
  - Sometimes all done "off-line", sometimes at run-time

What the Search Space Looks Like



A 2-D slice of a 3-D register-tile search space. The dark blue region was pruned. (Platform: Sun Ultra-IIi, 333 MHz, 667 Mflop/s peak, Sun cc v5.0 compiler)

ATLAS (DGEMM n = 500)



- ATLAS is faster than all other portable BLAS implementations and it is comparable with machine-specific libraries provided by the vendor.

What about more levels of memory?

- Need to minimize communication between all levels
  - Between L1 and L2 cache, cache and DRAM, DRAM and disk…
- The tiled algorithm requires finding a good block size
  - Machine dependent (cache aware – block size matched to hardware)
  - Need to "block" b x b matrix multiply in inner most loop
    - 1 level of memory $\Rightarrow$ 3 nested loops (naïve algorithm)
    - 2 levels of memory $\Rightarrow$ 6 nested loops
    - 3 levels of memory $\Rightarrow$ 9 nested loops …
- Cache Oblivious Algorithms offer an alternative
  - Treat nxn matrix multiply as a set of smaller problems
  - Eventually, these will fit in cache
  - Will minimize # words moved between every level of memory hierarchy – at least asymptotically
  - "Oblivious" to number and sizes of levels

Recursive Matrix Multiplication (RMM) (1/2)

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = A \cdot B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

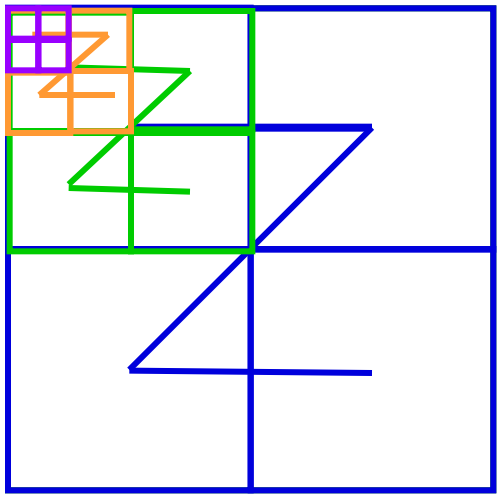
$$= \begin{pmatrix} A_{11}\cdot B_{11} + A_{12}\cdot B_{21} & A_{11}\cdot B_{12} + A_{12}\cdot B_{22} \\ A_{21}\cdot B_{11} + A_{22}\cdot B_{21} & A_{21}\cdot B_{12} + A_{22}\cdot B_{22} \end{pmatrix}$$

C_{11}	C_{12}	=	A_{11}	A_{12}	•	B_{11}	B_{12}	=	$A_{11}*B_{11} + A_{12}*B_{21}$	$A_{11}*B_{12} + A_{12}*B_{22}$
C_{21}	C_{22}		A_{21}	A_{22}		B_{21}	B_{22}		$A_{21}*B_{11} + A_{22}*B_{21}$	$A_{21}*B_{12} + A_{22}*B_{22}$

- Eventually, the matrices will fit in cache
- Don't need to optimized for size ☺
- But call overhead is high ☹

Recursive Data Layouts

- A related idea is to use a recursive structure for the matrix
  - Improve locality with machine-independent data structure
  - Can minimize latency with multiple levels of memory hierarchy
- There are several possible recursive decompositions depending on the order of the sub-blocks
- This figure shows Z-Morton Ordering ("space filling curve")
- See papers on "cache oblivious algorithms" and "recursive layouts"
  - Gustavson, Kagstrom, et al, SIAM Review, 2004



Advantages:
- the recursive layout works well for any cache size

Disadvantages:
- The index calculations to find A[i,j] are expensive
- Implementations switch to column-major for small sizes

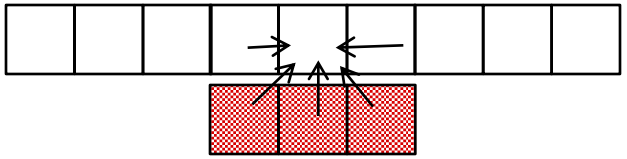
Optimizing in Practice

- Tiling for registers
  - loop unrolling, use of named "register" variables
- Tiling for multiple levels of cache and TLB
- Exploiting fine-grained parallelism in processor
  - Remove dependencies; SIMD
- Complicated compiler interactions (flags)
- Hard to do by hand (but you'll try)
- Automatic performance tuning (Berkeley centric):
  - ASPIRE: `aspire.eecs.berkeley.edu`
  - BeBOP: `bebop.cs.berkeley.edu`
  - PHiPAC: `www.icsi.berkeley.edu/~bilmes/phipac`

    in particular `tr-98-035.ps.gz`
  - ATLAS: `www.netlib.org/atlas`

Help Compiler Manage 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

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
- In Fortran, can use function calls (arguments assumed unaliased, maybe).

Unroll Loops

- 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;
```

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;
```

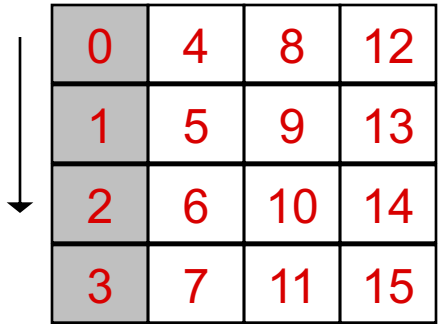
Pointer vs. array expression costs may differ.

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

Copy optimization

- Copy input operands or blocks
  - Reduce cache conflicts
  - Constant array offsets for fixed size blocks
  - Expose page-level locality
  - Alternative: use different data structures from start (if users willing)
    - Recall recursive data layouts

Original matrix
(numbers are addresses)



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

Reorganized into
2x2 blocks

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

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

Let $M = \begin{pmatrix} m11 & m12 \\ m21 & m22 \end{pmatrix} = \begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix} \begin{pmatrix} b11 & b12 \\ b21 & b22 \end{pmatrix}$

Let p1 = (a12 - a22) * (b21 + b22)

p2 = (a11 + a22) * (b11 + b22)

p3 = (a11 - a21) * (b11 + b12)

p4 = (a11 + a12) * b22

p5 = a11 * (b12 - b22)

p6 = a22 * (b21 - b11)

p7 = (a21 + a22) * b11

Then m11 = p1 + p2 - p4 + p6

m12 = p4 + p5

m21 = p6 + p7

m22 = p2 - p3 + p5 - p7

Extends to nxn by divide&conquer

Strassen (continued)

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

- 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

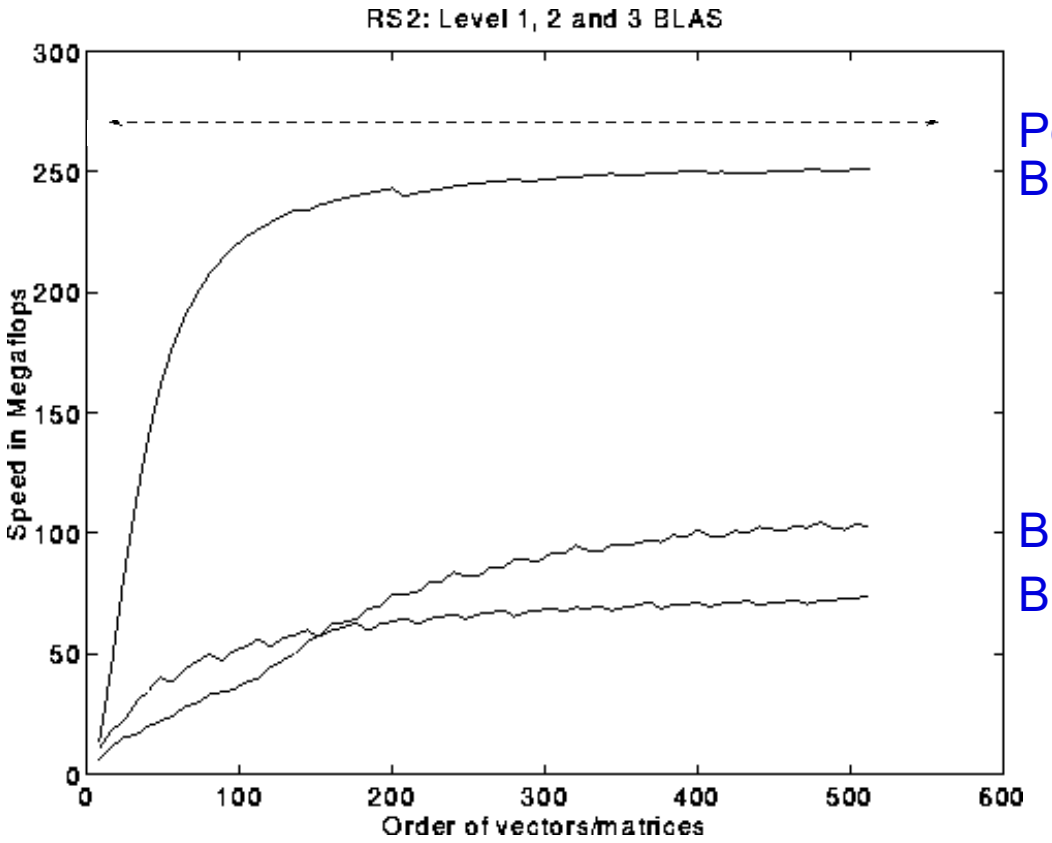
- World's record was $O(n^{2.37548...})$
  - Coppersmith & Winograd, 1987
- New Record! 2.37*548* reduced to 2.37*293*
  - Virginia Vassilevska Williams, UC Berkeley & Stanford, 2011
- Newer Record! 2.372*93* reduced to 2.372*86*
  - Francois Le Gall, 2014
- Lower bound on #words moved can be extended to (some) of these algorithms (2015 thesis of Jacob Scott)
- Possibility of $O(n^{2+\varepsilon})$ algorithm!
  - Cohn, Umans, Kleinberg, 2003
- Can show they all can be made numerically stable
  - Demmel, Dumitriu, Holtz, Kleinberg, 2007
- Can do rest of linear algebra (solve $Ax=b$, $Ax=\lambda x$, etc) as fast , and numerically stably
  - Demmel, Dumitriu, Holtz, 2008
- Fast methods (besides Strassen) may need unrealistically large n

Basic Linear Algebra Subroutines (BLAS)

- Industry standard interface (evolving)
  - www.netlib.org/blas, www.netlib.org/blas/blast--forum
- Vendors, others supply optimized implementations
- History
  - BLAS1 (1970s): 15 different operations
    - vector operations: dot product, saxpy (y=α*x+y), root-sum-squared, etc
    - m=2*n, f=2*n, q = f/m = computational intensity ~1 or less
  - BLAS2 (mid 1980s): 25 different operations
    - matrix-vector operations: matrix vector multiply, etc
    - m=n^2, f=2*n^2, q~2, less overhead
    - somewhat faster than BLAS1
  - BLAS3 (late 1980s): 9 different operations
    - matrix-matrix operations: matrix matrix multiply, etc
    - m <= 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 use BLAS3 when possible (LAPACK & ScaLAPACK)
  - See `www.netlib.org/{lapack,scalapack}`
  - More later in the course

BLAS speeds on an IBM RS6000/590

Peak speed = 266 Mflops



Peak
BLAS 3

BLAS 2

BLAS 1

Matrices start in DRAM Memory

BLAS 3 (n-by-n matrix matrix multiply) vs
BLAS 2 (n-by-n matrix vector multiply) vs
BLAS 1 (saxpy of n vectors)

Some reading for today

- 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 Adolfy 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

Summary of Lecture 2

- 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

Class Logistics

- Make sure you're set up in bcourses
  - https://bcourses.berkeley.edu/courses/1480197/
- Piazza:
  - piazza.com/berkeley/spring2019/cs267
- Homework 0 and 1 posted on web site
  - HW0 due Wednesday Jan 30 (9am)
  - HW1 online, but will have assigned teams
- Fill in on-line class survey by midnight tonight
  - We need this to assign teams for Homework 1