CS 267: Optimizing Matrix Multiply (part 2) and Introduction to Data Parallelism Lecture 3

Kathy Yelick

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

Outline

Matrix multiply (continued)

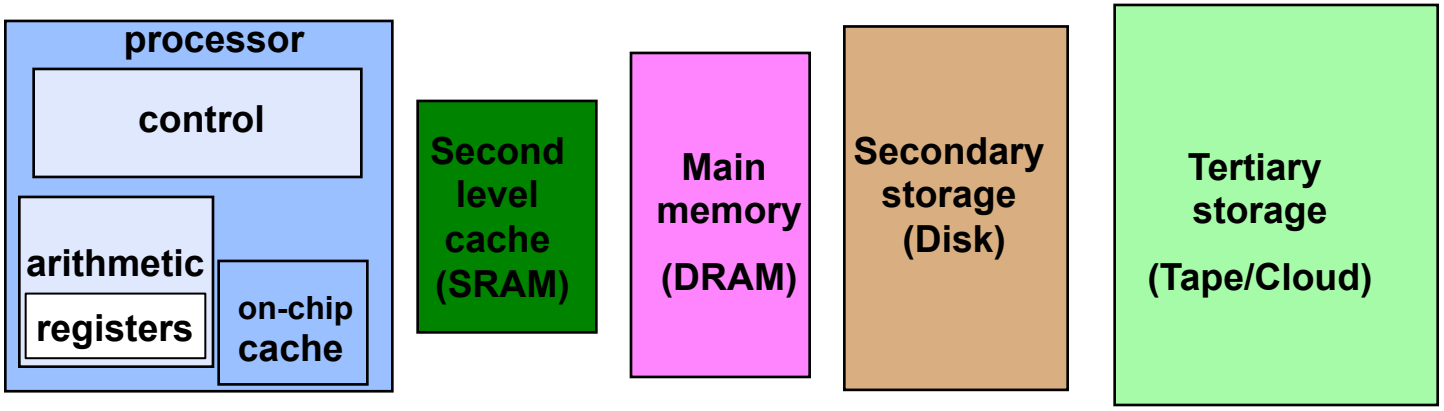
- Lecture 2 recap: memory hierarchies and tiling
- Cache Oblivious algorithms
- Practical guide to optimizations
- Beyond $O(n^3)$ matrix multiply

Introduction to parallel machines and programming

- Shared memory
- Distributed memory
- Data parallel

Lecture 2 Recap

- **Memory hierarchy exploits locality to improve average case performance**
  - **spatial locality:** accessing things nearby previous accesses
  - **temporal locality:** reusing an item that was previously accessed



- **Roofline model:** upper limit of bandwidth and compute speed (more on this next week)
- **Tiling:** adding nested loops to improve temporal locality
- **Communication lower bound:** minimum data movement necessary for the algorithm

Blocked (Tiled) Matrix Multiply

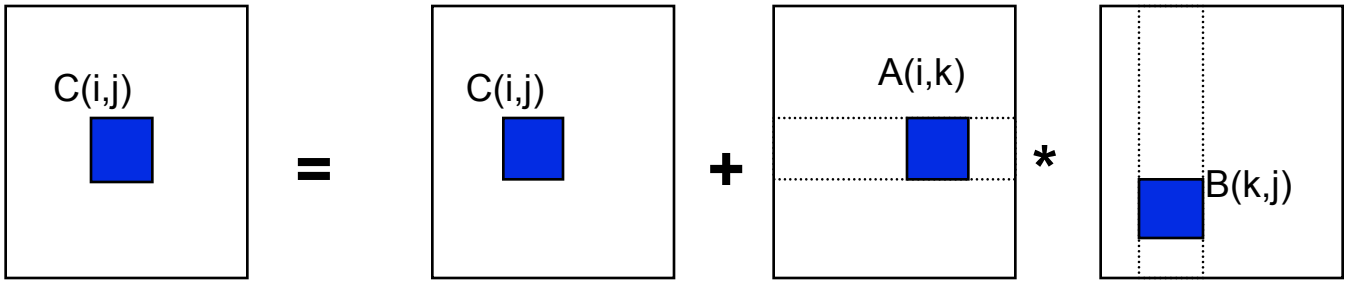
Consider A,B,C to be N-by-N matrices of b-by-b subblocks where
b=n / N is called the block size

```
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)

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 $q = O(\sqrt{M})$

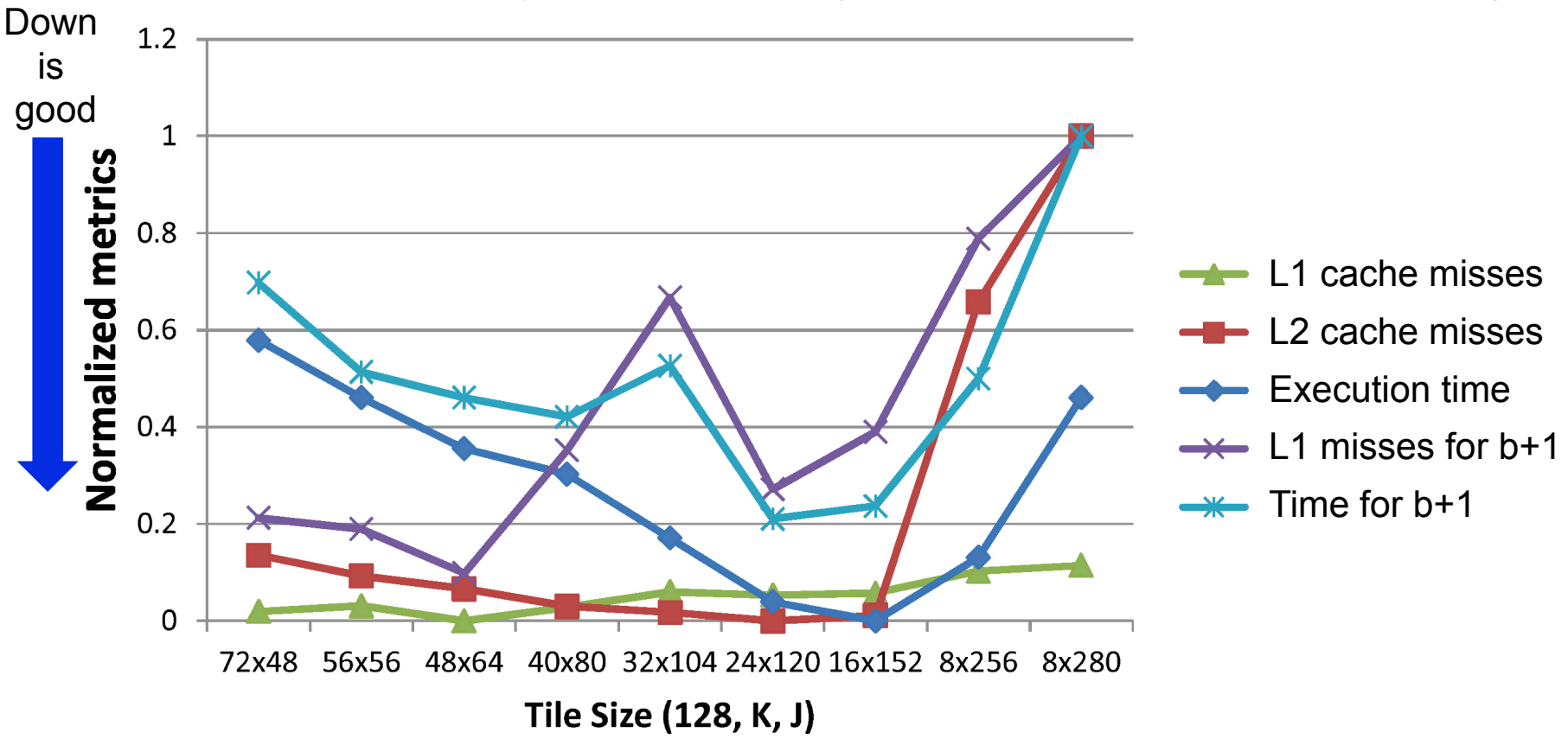
Data transferred min:

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

- 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)

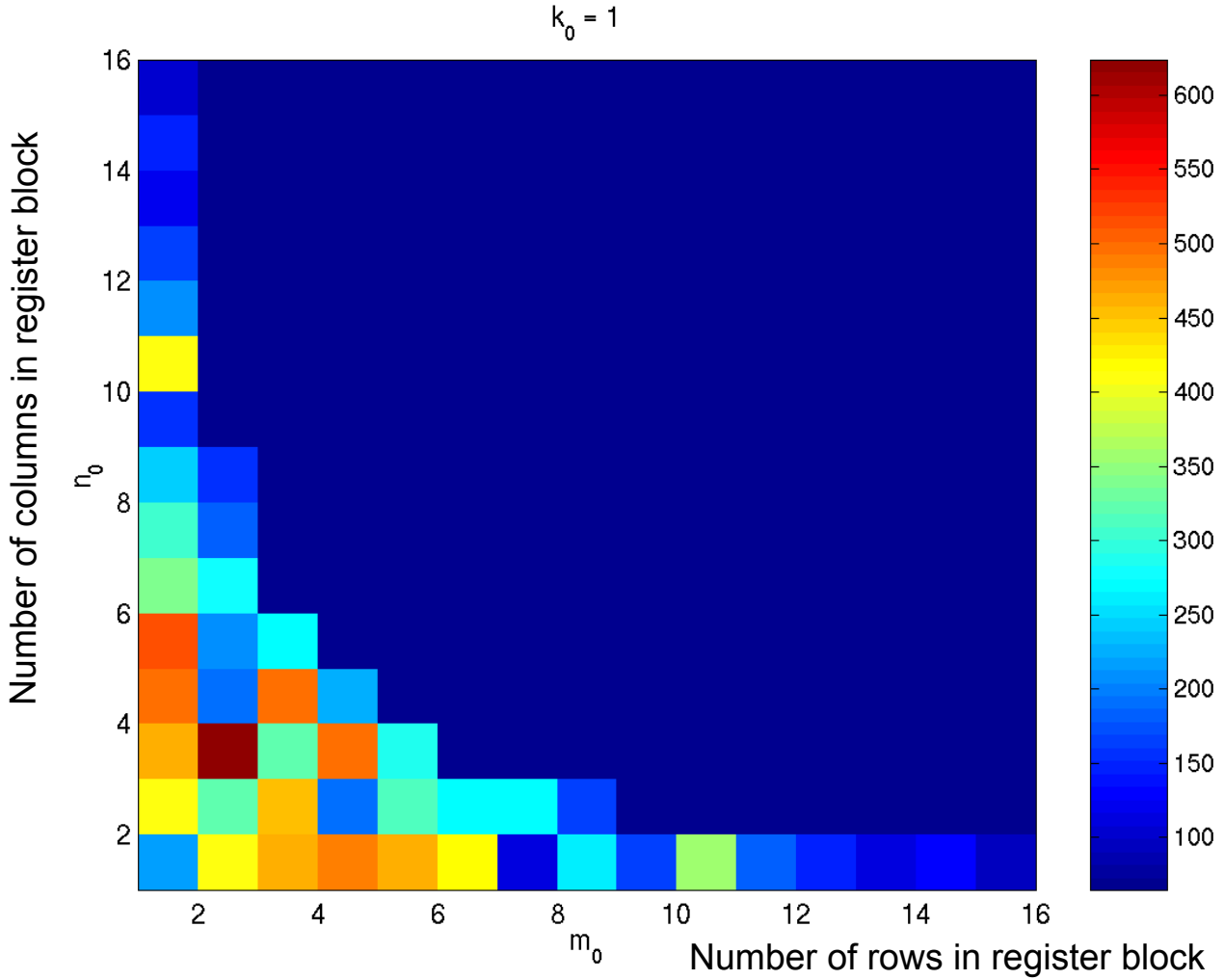


Hundreds of papers on this; above from [Mehtra, Beeraka, Yew, 2013]

Tuning Code in Practice

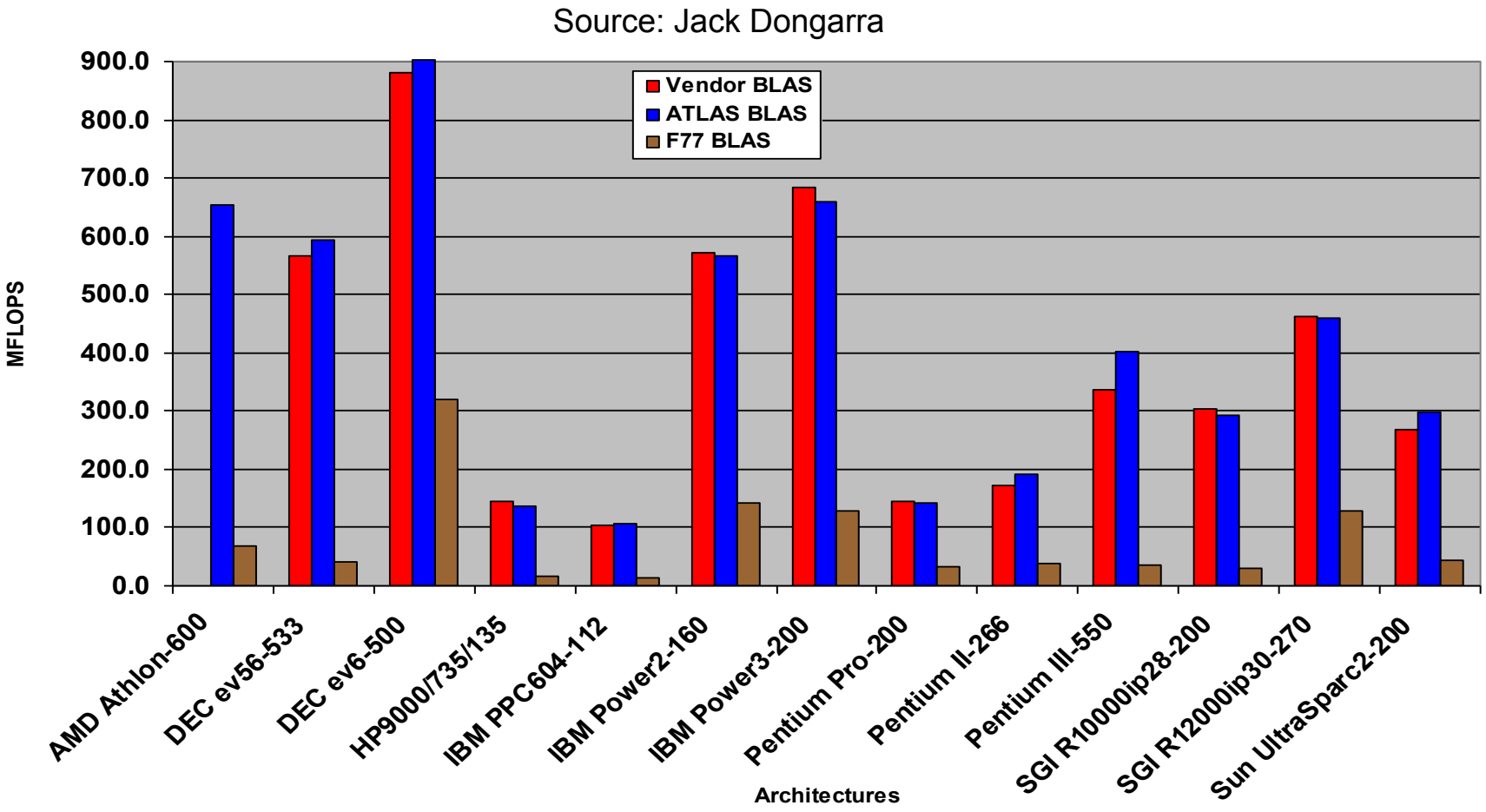
- Tuning code can be tedious
  - Many optimizations besides blocking
  - Behavior of machine and compiler hard to predict
- Approach #1: Analytical performance models
  - Use model (of cache, memory costs, etc.) to select best code
  - But model needs to be both simple and accurate ☹
- Approach #2: "Autotuning"
  - 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 move 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 ⇒ 3 nested loops (naïve algorithm)
    - 2 levels of memory ⇒ 6 nested loops
    - 3 levels of memory ⇒ 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}
C_{21}	C_{22}

$=$

A_{11}	A_{12}
A_{21}	A_{22}

$\bullet$

B_{11}	B_{12}
B_{21}	B_{22}

$=$

$A_{11*}B_{11} + A_{12*}B_{21}$	$A_{11*}B_{12} + A_{12*}B_{22}$
$A_{21*}B_{11} + A_{22*}B_{21}$	$A_{21*}B_{12} + A_{22*}B_{22}$

- True when each bock is a 1x1 or n/2 x n/2
- For simplicity: square matrices with $n = 2^m$
  - Extends to general rectangular case

Recursive Matrix Multiplication (2/2)

```
func C = RMM (A, B, n)
    if n=1, C = A * B, else
      {  C11 = RMM (A11 , B11 , n/2) + RMM (A12 , B21 , n/2)
         C12 = RMM (A11 , B12 , n/2) + RMM (A12 , B22 , n/2)
         C21 = RMM (A21 , B11 , n/2) + RMM (A22 , B21 , n/2)
         C22 = RMM (A21 , B12 , n/2) + RMM (A22 , B22 , n/2)  }
    return
```

A(n) = # arithmetic operations in RMM(. , . , n)
= $8 \cdot A(n/2) + 4(n/2)^2$ if $n > 1$, else 1
= $2n^3$ … same operations as usual, in different order

W(n) = # words moved between fast, slow memory by RMM(. , . , n)
= $8 \cdot W(n/2) + 4 \cdot 3(n/2)^2$ if $3n^2 > M_{fast}$, else $3n^2$
= $O(n^3 / (M_{fast})^{1/2} + n^2)$ … same as blocked matmul
Don't need to know M_{fast} for this to work!

Recursion: Cache Oblivious Algorithms

- The tiled algorithm requires finding a good block size
- Cache Oblivious Algorithms offer an alternative
  - Treat nxn matrix multiply set of smaller problems
  - Eventually, these will fit in cache
- Cases for A (nxm) * B (mxp)
  - Case1: n>= max{m,p}: split A horizontally:
  - Case 2: m>= max{n,p}: split A vertically and B horizontally
  - Case 3: p>= max{m,n}: split B vertically

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}$$

Case 1

$$(A_1, A_2)\begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = (A_1 B_1 + A_2 B_2)$$

Case 2

$$A(B_1, B_2) = (A\,B_1, A\,B_2)$$

Case 3

- Attains lower bound in O() sense

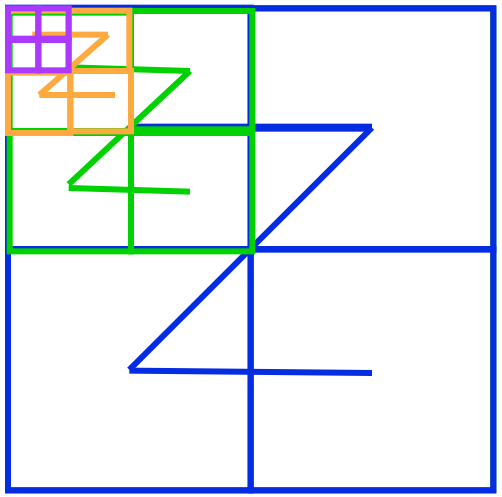
Experience with Cache-Oblivious Algorithms

- In practice, need to cut off recursion well before 1x1 blocks
  - Call “micro-kernel” on small blocks
  - Careful attention to micro-kernel is needed
- Pingali et al report that they never got more than 2/3 of peak.
  - Fully recursive approach with highly optimized recursive micro-kernel
  - (unpublished, presented at LACSI’06)

- Issues with Cache Oblivious (recursive) approach

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
  - 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/phipac`

    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]

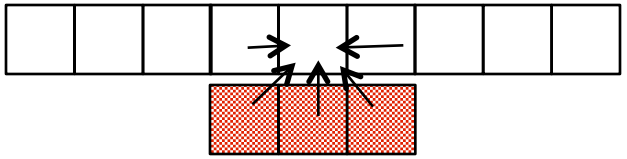
```
float f1 = b[i];
float f2 = b[i+1];

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

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

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;
```

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;
```

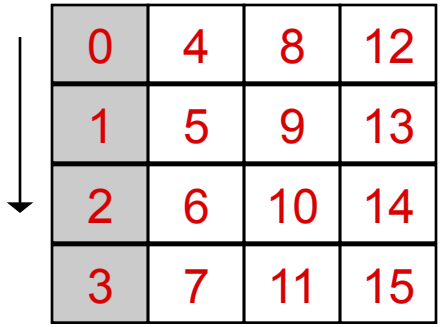
Pointer vs. array expression costs may differ.

- 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

Locality in Other Algorithms

- The performance of any algorithm is limited by q
  - q = "computational intensity" = #arithmetic_ops / #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

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
    $= \Omega(n^{\log2\ 7} / M^{(\log2\ 7)/2 - 1}) \sim \Omega(n^{2.81} / M^{0.4})$
    (Ballard, D., Holtz, Schwartz, 2011, **SPAA Best Paper Prize**)
  - Attainable too, more on parallel version later

Other Fast Matrix Multiplication Algorithms

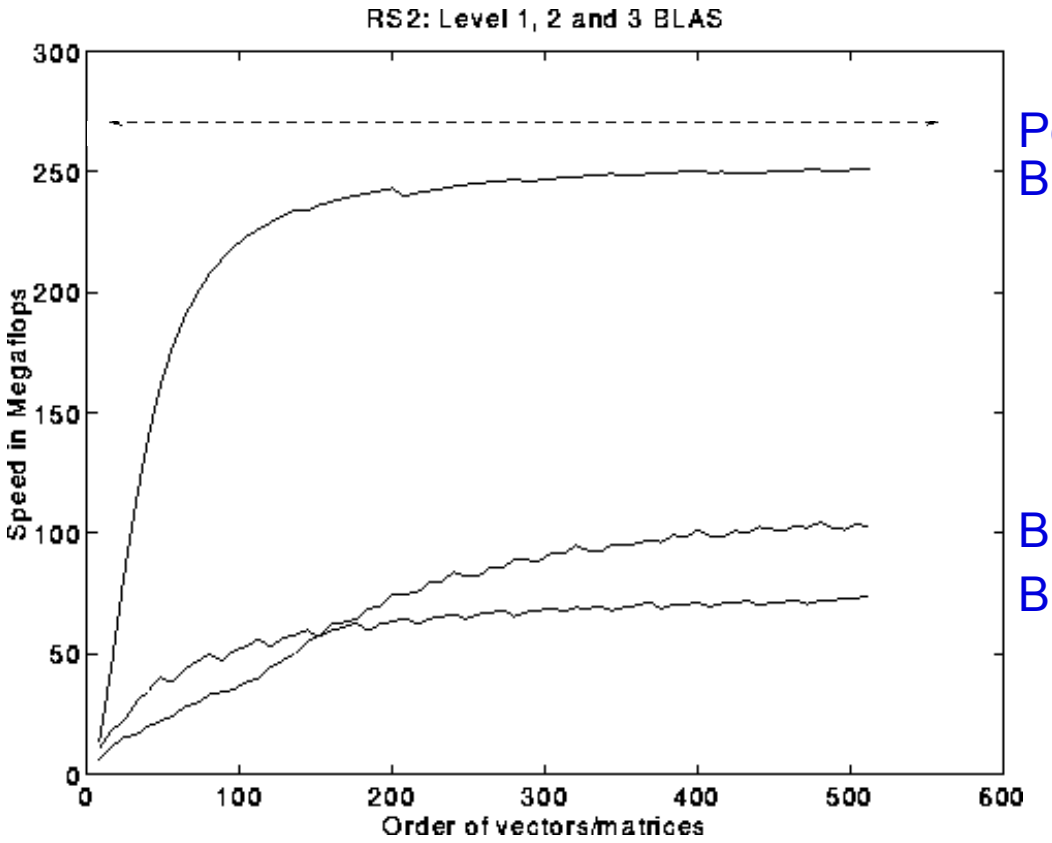
- World's record was $O(n^{2.37548...})$
  - Coppersmith & Winograd, 1987
- New Record! 2.37*548* reduced to 2.37*293*
  - Virginia Vassilevska Williams, UC Berkeley & Stanford, 2011
- Newer Record! 2.372*93* reduced to 2.372*86*
  - Francois Le Gall, 2014
- Lower bound on #words moved can be extended 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

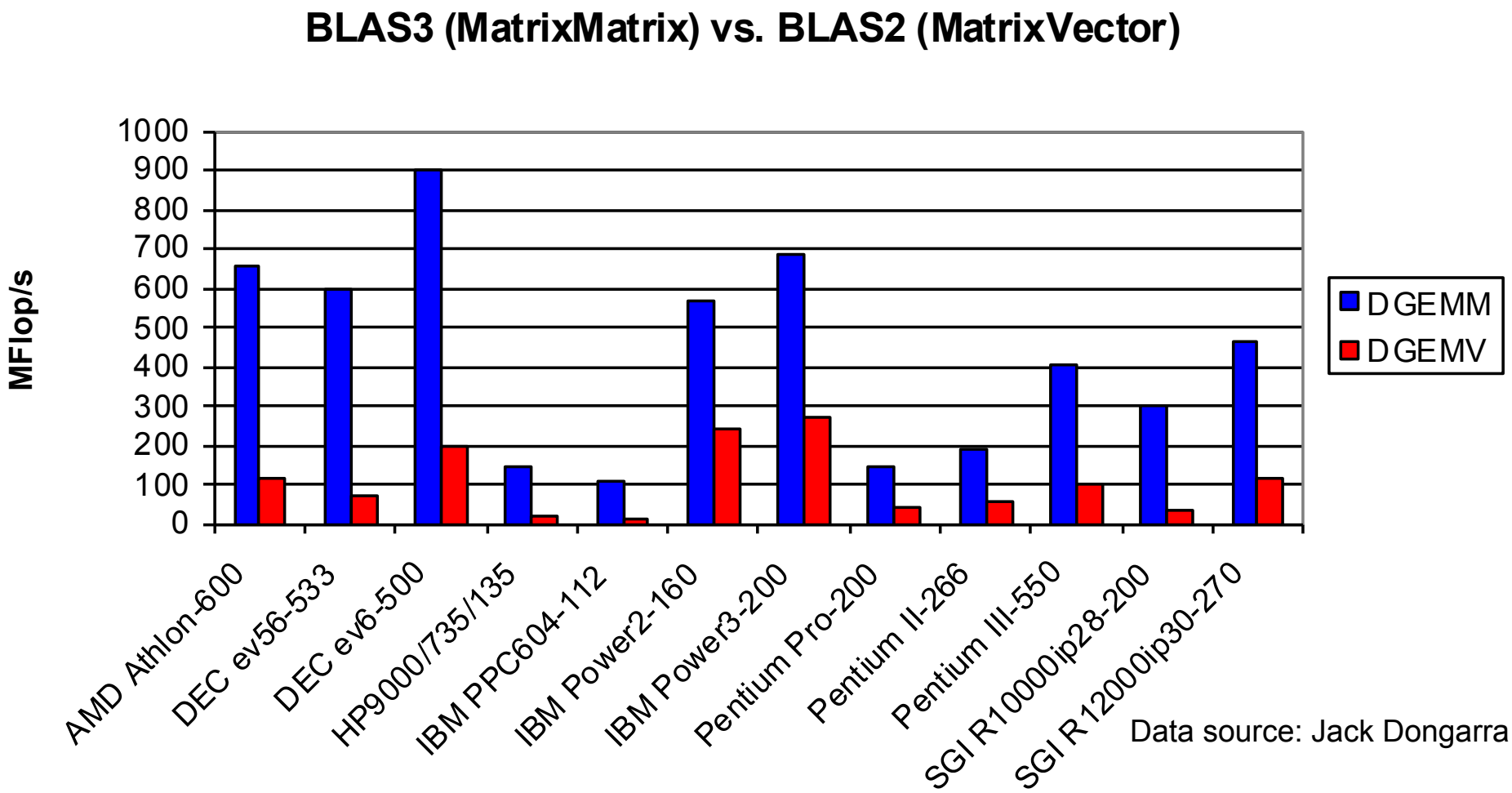
BLAS 1

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)

Dense Linear Algebra: BLAS2 vs. BLAS3

- BLAS2 and BLAS3 have very different computational intensity, and therefore different performance



Data source: Jack Dongarra

Measuring Performance — Runtime

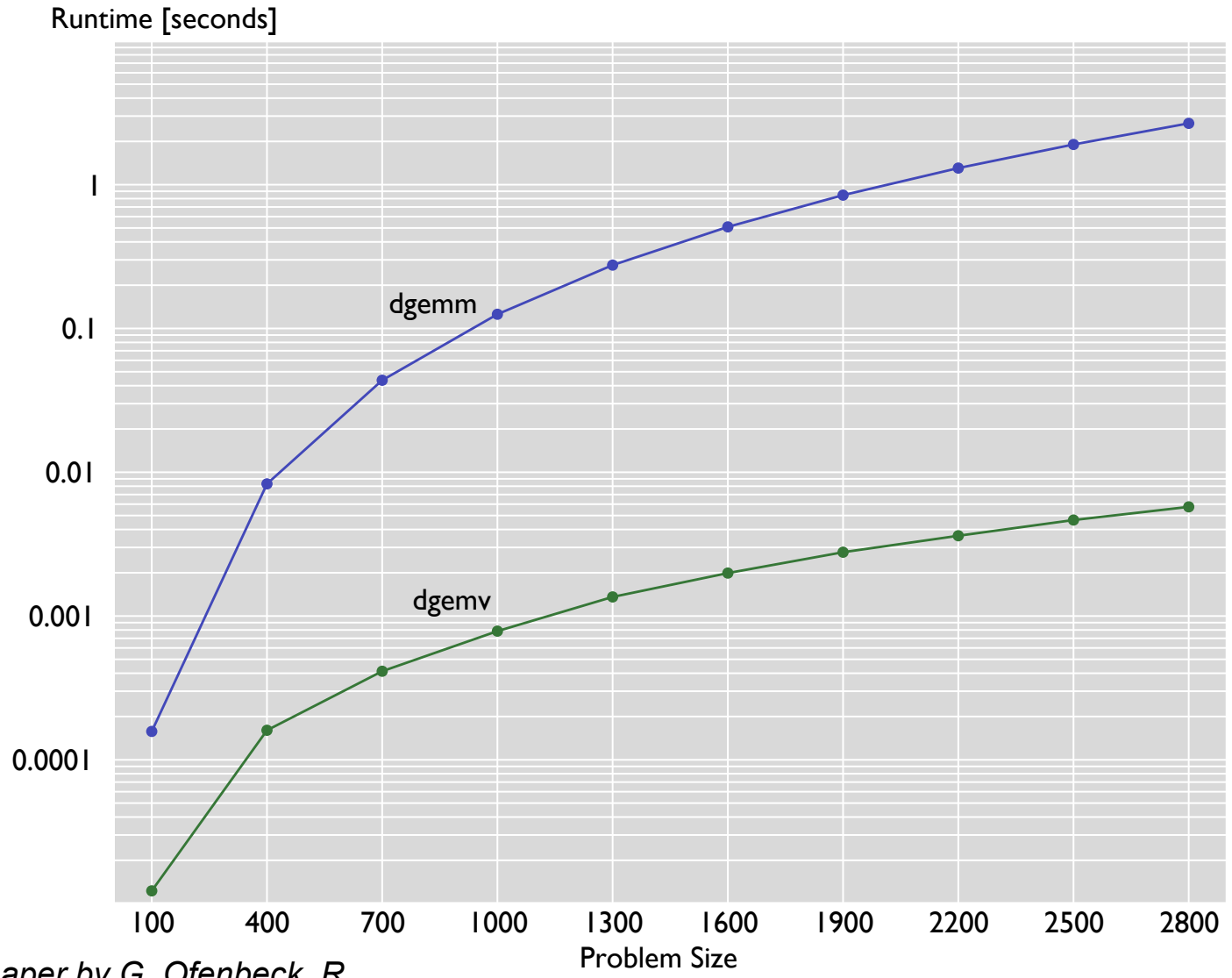


Image and paper by G. Ofenbeck, R. Steinman, V. Caparrós Cabezas, D. Spampinato, M. Püschel

Measuring Performance — Flops/Cycle

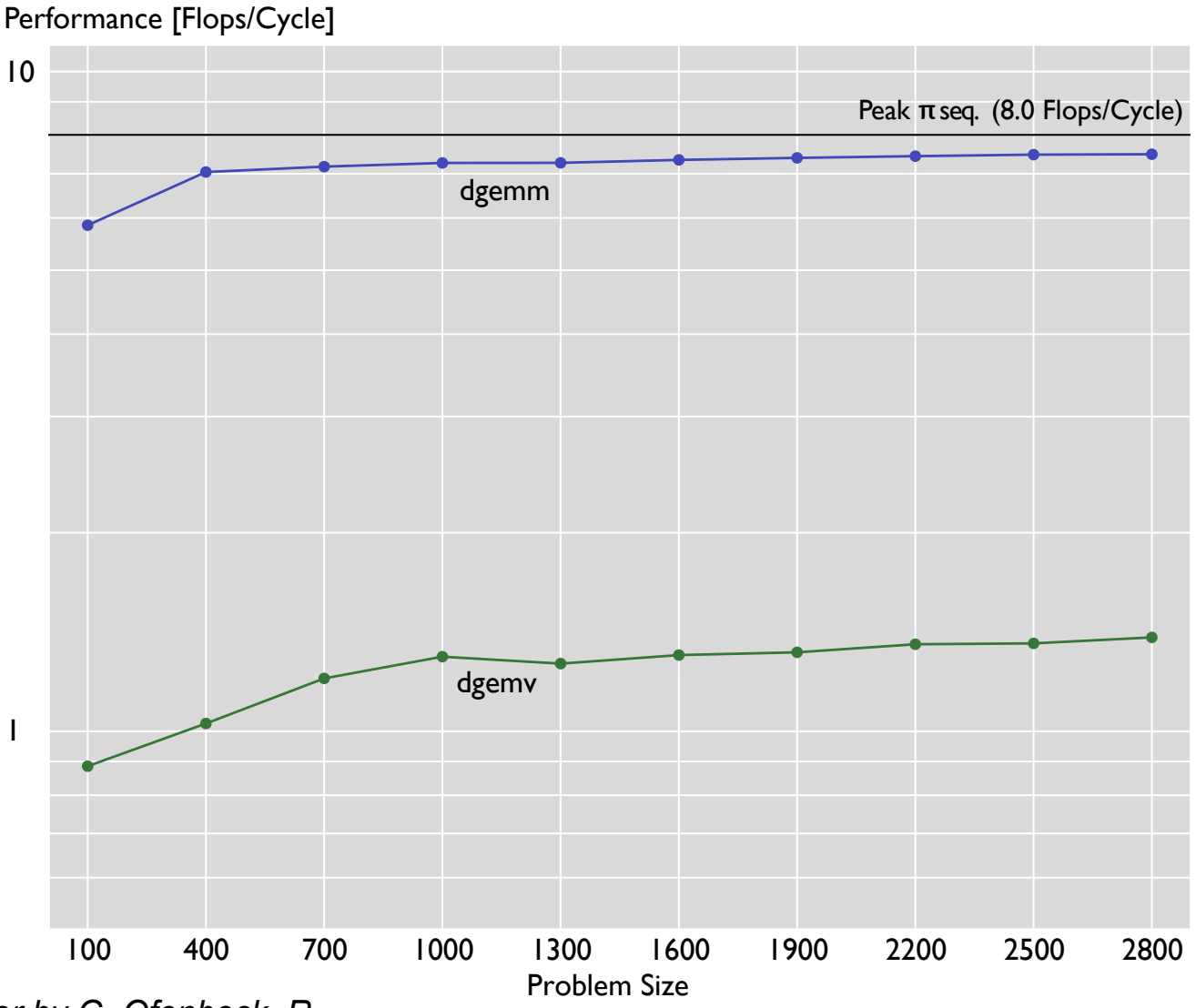


Image and paper by G. Ofenbeck, R. Steinman, V. Caparrós Cabezas, D. Spampinato, M. Püschel

Measuring Performance — Roofline Plot

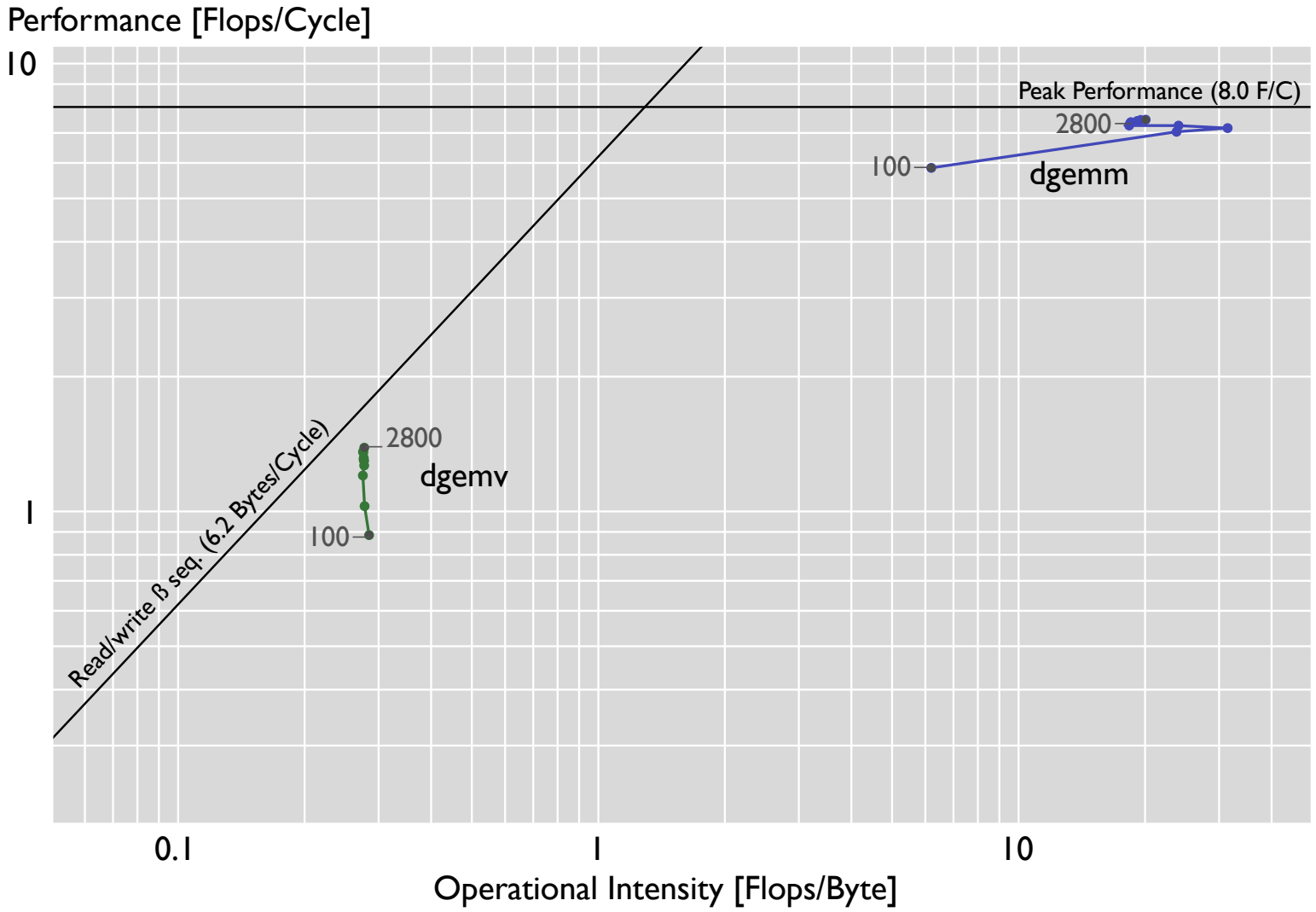


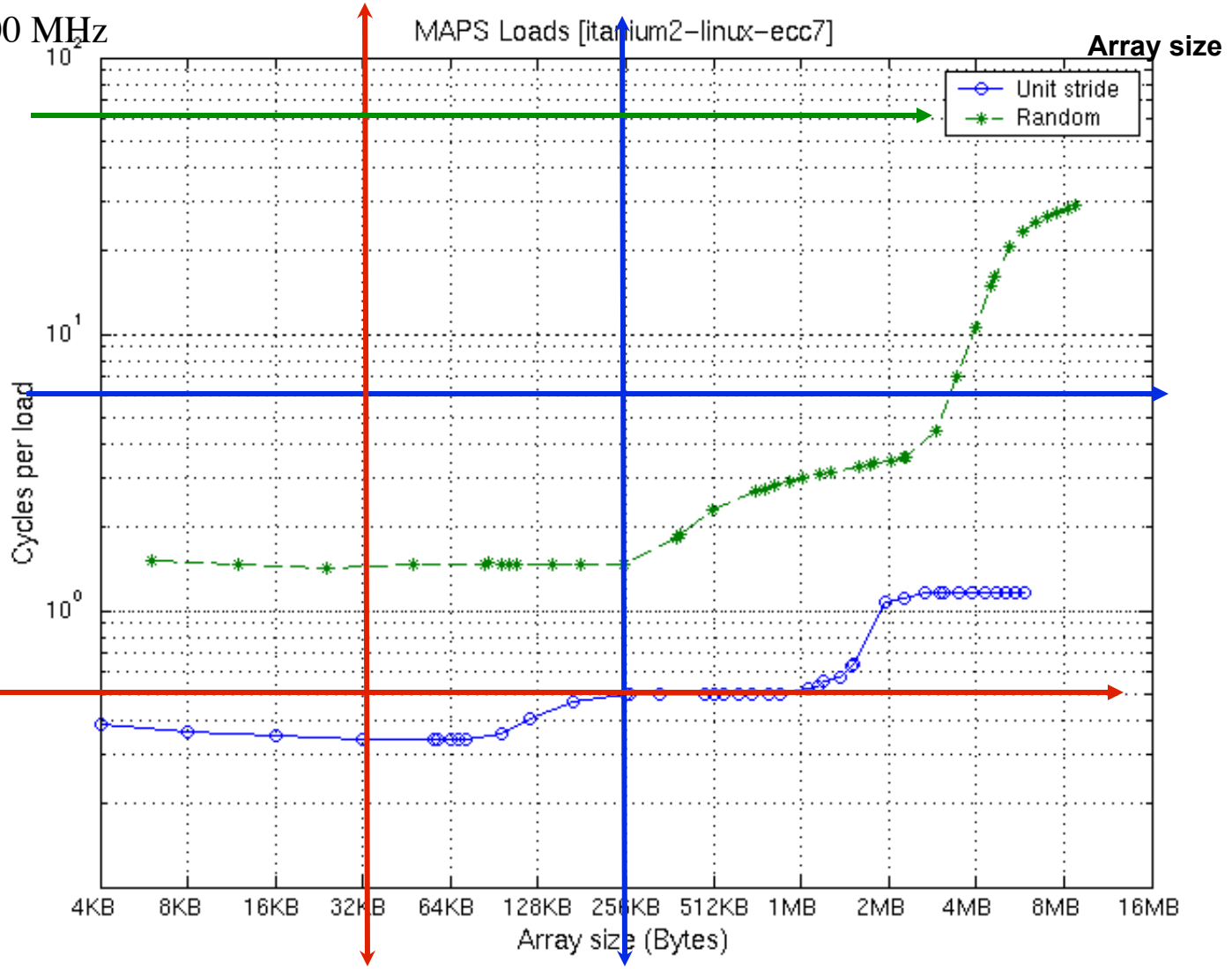
Image and paper by G. Ofenbeck, R. Steinman, V. Caparrós Cabezas, D. Spampinato, M. Püschel

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

Memory Performance on Itanium 2 (CITRIS)

Itanium2, 900 MHz



Mem:
11-60 cycles

L3: 2 MB
128 B line
3-20 cycles

L2: 256 KB
128 B line
.5-4 cycles

L1: 32 KB
64B line
.34-1 cycles

Uses MAPS Benchmark: http://www.sdsc.edu/PMaC/MAPs/maps.html

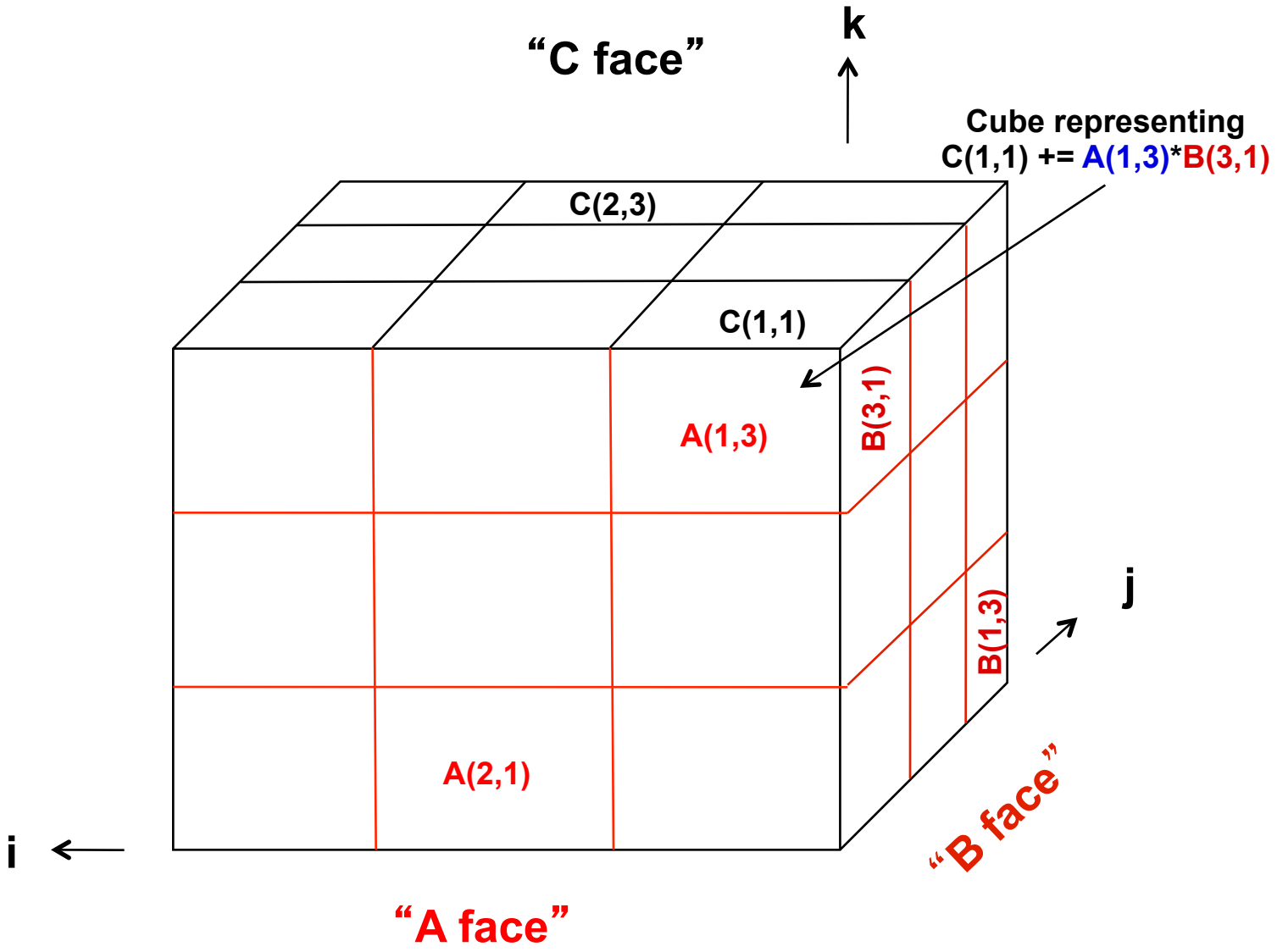
Proof of Communication Lower Bound on C = A*B (1/6)

- Proof from Irony/Tishkin/Toledo (2004)
  - We'll need it for the communication lower bound on parallel matmul
- Think of instruction stream being executed
  - Looks like " … add, load, multiply, store, load, add, …"
  - We want to count the number of loads and stores, given that we are multiplying n-by-n matrices C = A*B using the usual $2*n^3$ flops, possibly reordered assuming addition is commutative/associative
    - It actually isn't associative in floating point, but close enough
  - Assuming that at most M words can be stored in fast memory
- Outline:
  - Break instruction stream into segments, each containing M loads and stores
  - Somehow bound the maximum number of adds and multiplies that could be done in each segment, call it F
  - So F · # segments $\geq 2 \cdot n^3$, and # segments $\geq 2 \cdot n^3 / F$
  - So # loads & stores = M · #segments $\geq M \cdot 2 \cdot n^3 / F$

Proof of Communication Lower Bound on C = A*B (2/6)

- Given segment of instruction stream with M loads & stores, how many adds & multiplies (F) can we do?
  - At most 2M entries of C, 2M entries of A and/or 2M entries of B can be accessed
- Use geometry:
  - Represent $2 \cdot n^3$ operations by n x n x n cube
  - One n x n face represents A
    - each 1 x 1 subsquare represents one A(i,k)
  - One n x n face represents B
    - each 1 x 1 subsquare represents one B(k,j)
  - One n x n face represents C
    - each 1 x 1 subsquare represents one C(i,j)
  - Each 1 x 1 x 1 subcube represents one C(i,j) += A(i,k) * B(k,j)

Proof of Communication Lower Bound on C = A*B (3/6)

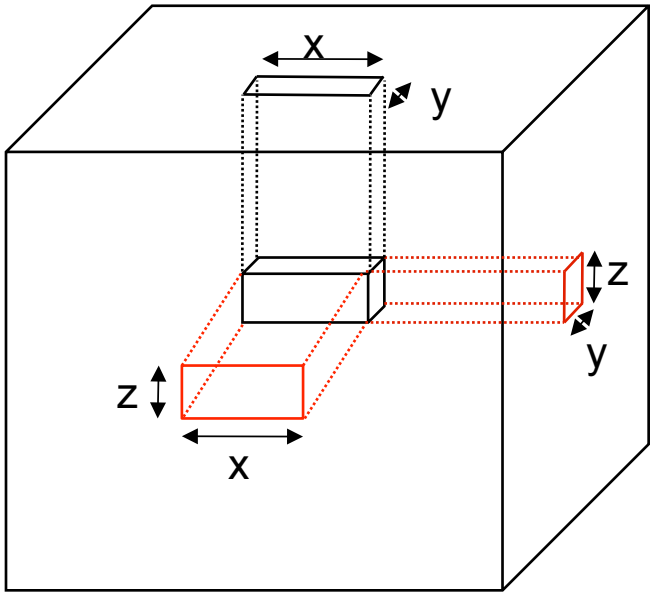


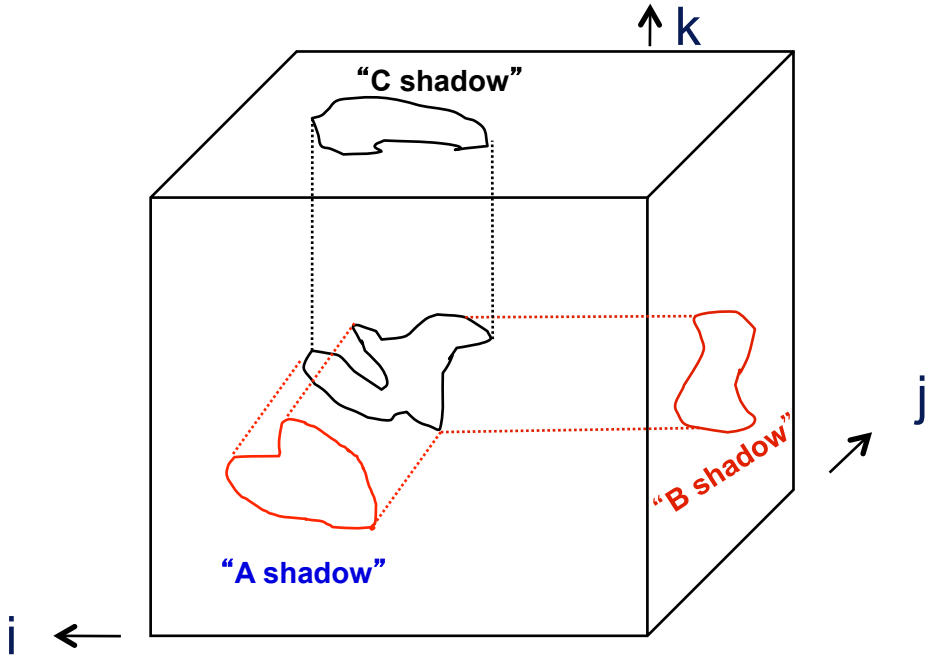
Proof of Communication Lower Bound on C = A*B (4/6)

- Given segment of instruction stream with M load & stores, how many adds & multiplies (F) can we do?
  - At most 2M entries of C, and/or of A and/or of B can be accessed
- Use geometry:
  - Represent $2 \cdot n^3$ operations by n x n x n cube
  - One n x n face represents A
    - each 1 x 1 subsquare represents one A(i,k)
  - One n x n face represents B
    - each 1 x 1 subsquare represents one B(k,j)
  - One n x n face represents C
    - each 1 x 1 subsquare represents one C(i,j)
  - Each 1 x 1 x 1 subcube represents one C(i,j) += A(i,k) * B(k,j)

- If we have at most 2M "A squares", 2M "B squares", and 2M "C squares" on faces, how many cubes can we have?

Proof of Communication Lower Bound on C = A*B (5/6)





cubes in black box with side lengths x, y and z
= Volume of black box
= x*y*z
= (#A□s * #B□s * #C□s)$^{1/2}$
= (xz * zy * yx)$^{1/2}$

(i,k) is in "A shadow" if (i,j,k) in 3D set
(j,k) is in "B shadow" if (i,j,k) in 3D set
(i,j) is in "C shadow" if (i,j,k) in 3D set

Thm (Loomis & Whitney, 1949)
cubes in 3D set = Volume of 3D set
≤ (area(A shadow) * area(B shadow) * area(C shadow)) $^{1/2}$

Proof of Communication Lower Bound on C = A*B (6/6)

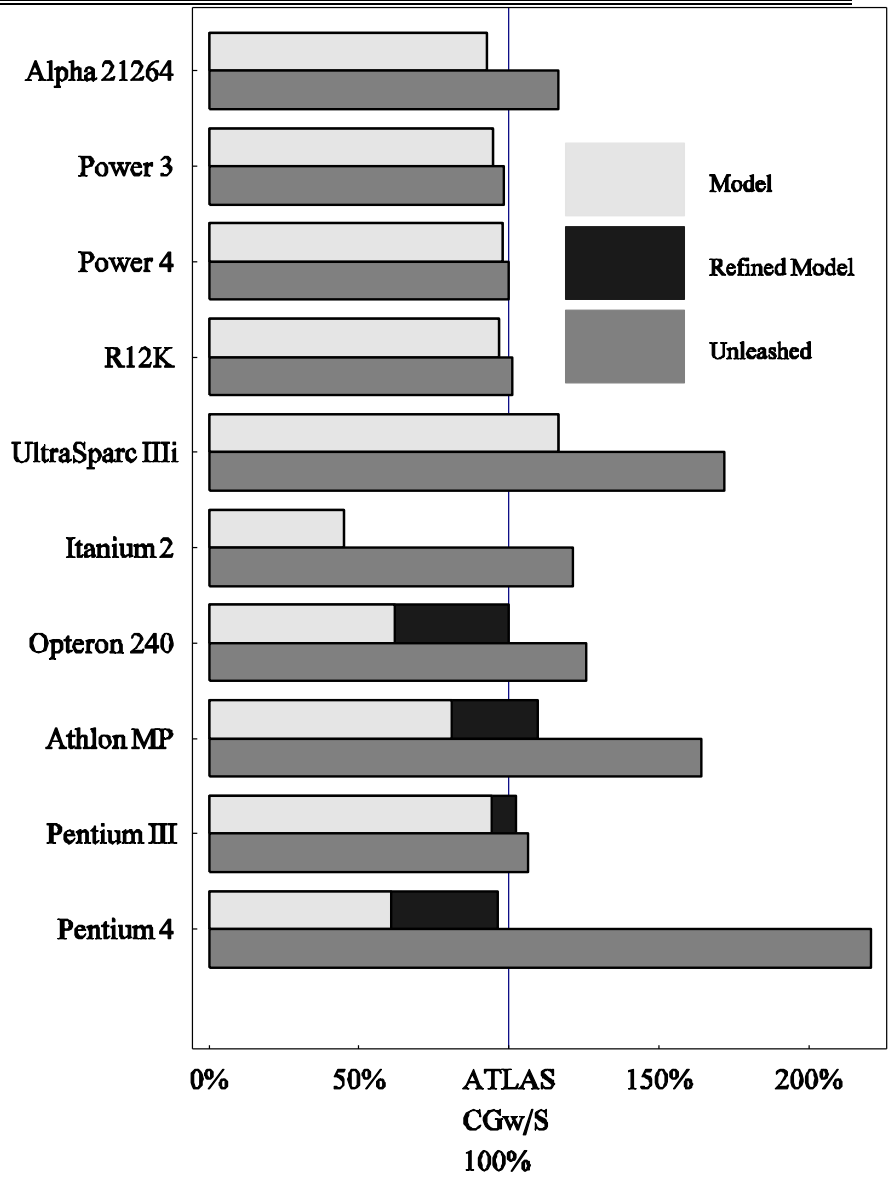
- Consider one "segment" of instructions with M loads and stores
- There can be at most 2M entries of A, B, C available in one segment
- Volume of set of cubes representing possible multiply/adds ≤ $(2M \cdot 2M \cdot 2M)^{1/2} = (2M)^{3/2} \equiv F$
- # Segments ≥ $2n^3 / F$
- # Loads & Stores = M · #Segments ≥ $M \cdot 2n^3 / F = n^3 / (2M)^{1/2}$

- Parallel Case: apply reasoning to one processor out of P
  - # Adds and Muls = $2n^3 / P$ (assuming load balanced)
  - M= n^2 / P (each processor gets equal fraction of matrix)
  - # "Load & Stores" = # words communicated with other procs ≥ $M \cdot (2n^3 /P) / F = M \cdot (2n^3 /P) / (2M)^{3/2} = n^2 / (2P)^{1/2}$

Experiments on Search vs. Modeling

Study compares search (Atlas) to optimization selection based on performance models

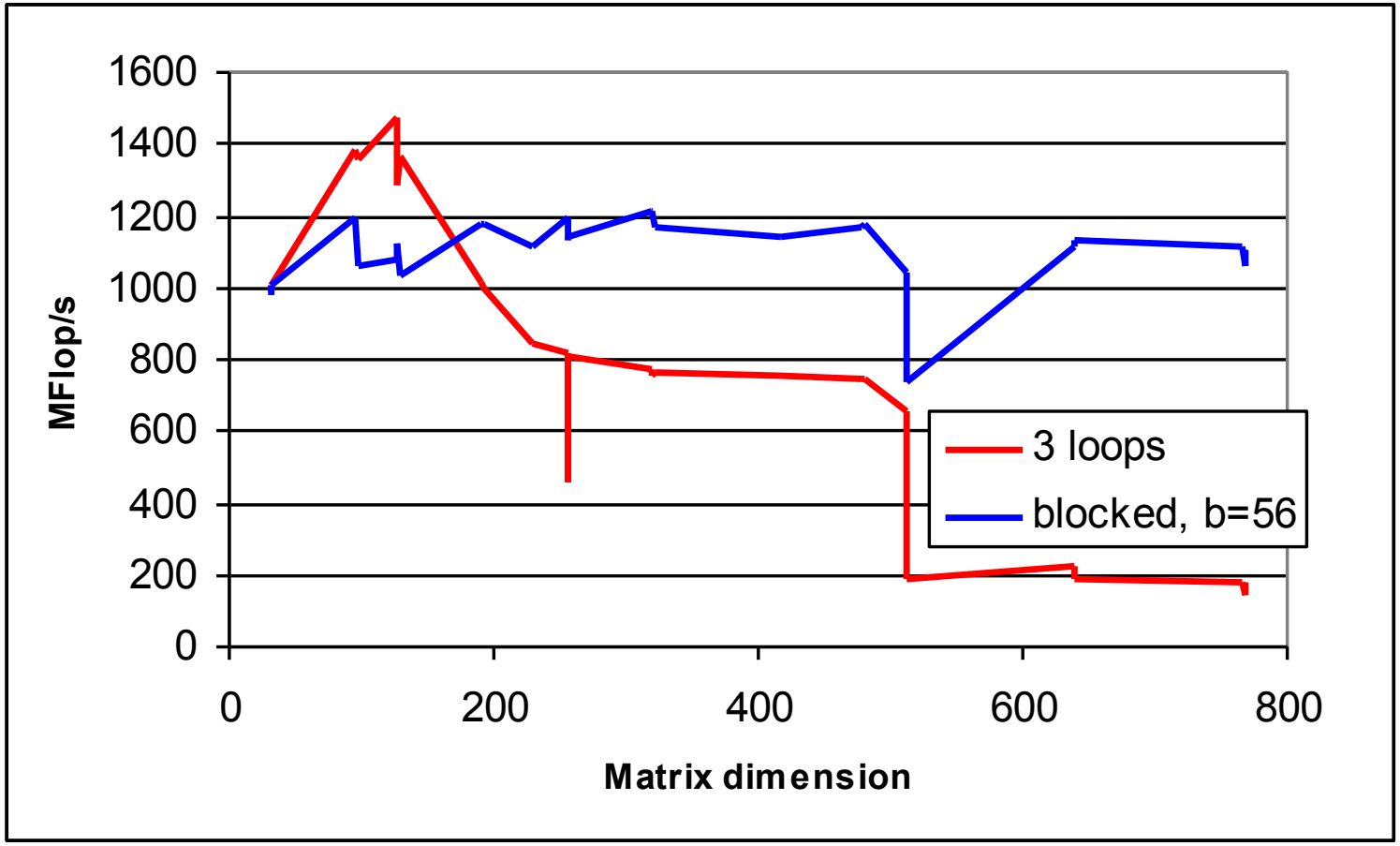
- Ten modern architectures
- Model did well on most cases
  - Better on UltraSparc
  - Worse on Itanium
- Eliminating performance gaps: think globally, search locally
  - -small performance gaps: local search
  - -large performance gaps: refine model
- Substantial gap between ATLAS CGw/S and ATLAS Unleashed on some machines



Source: K. Pingali. Results from IEEE '05 paper by K Yotov, X Li, G Ren, M Garzarán, D Padua, K Pingali, P Stodghill.

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



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 = # floating point operations / # slow memory references

  2. machine balance in the memory system:
     - t_m/t_f = time for slow memory reference / 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
- Now it's your turn: Homework 1
  - Work in teams of 2 or 3 (assigned this time)

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?

Outline

Matrix multiply (continued)

✓Lecture 2 recap: memory hierarchies and tiling

✓Cache Oblivious algorithms

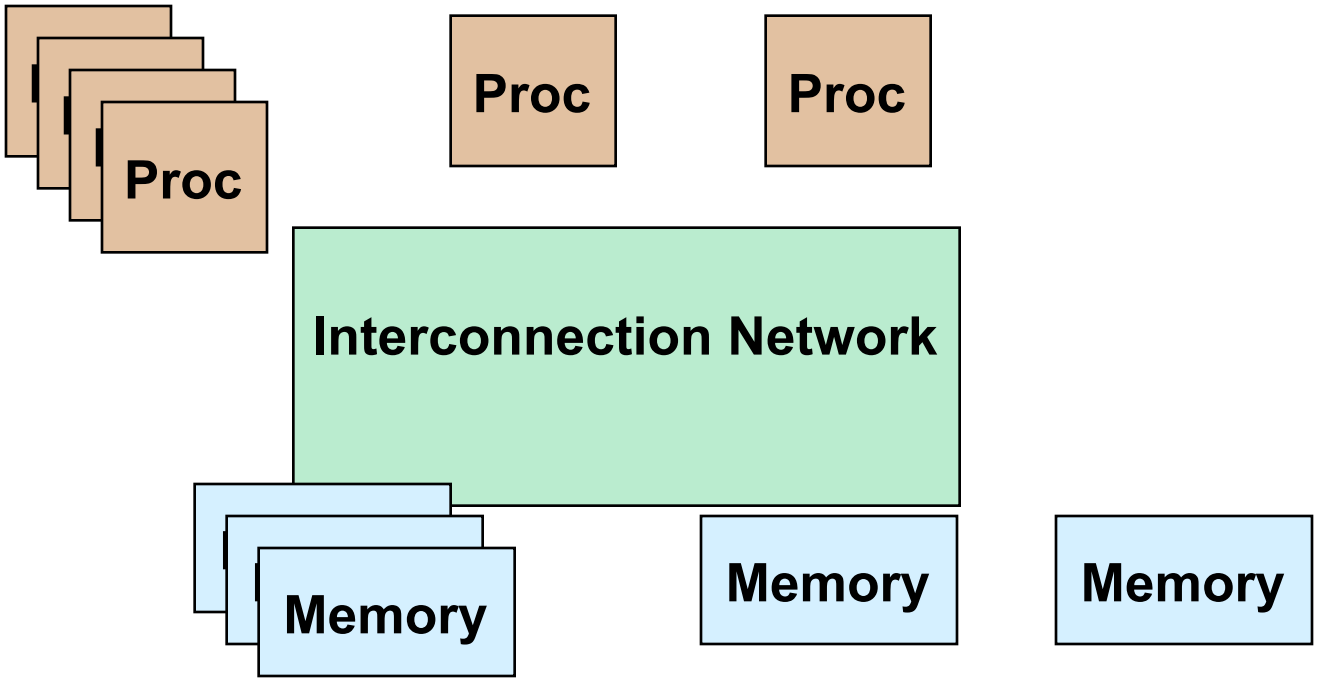
✓Practical guide to optimizations

✓Beyond $O(n^3)$ matrix multiply

Introduction to parallel machines and programming

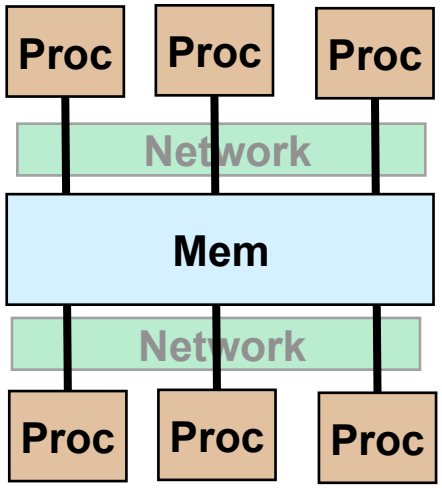
- Shared memory
- Distributed memory
- Data parallel

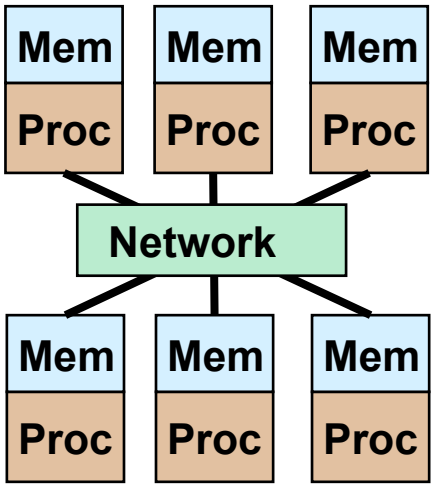
A generic parallel architecture

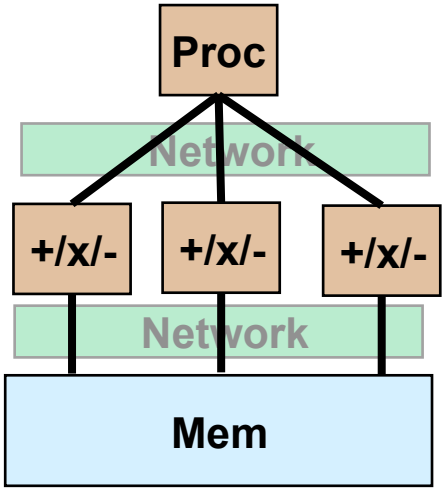


- Where is the memory physically located?
- Is it connected directly to processors?
- What is the connectivity of the network?
- How are the processor controlled?

Parallel Machines and Programming







Shared Memory	Distributed Memory	Single Instruction Multiple Data (SIMD)
Processors execute own instruction stream	Processors execute own instruction stream	One instruction stream (all run same instruction)
Communicate by reading/writing memory	Communicate by sending messages	Communicate through memory
Cost of a read/write is constant	Message time depends on size, but not location	Assume unbounded # of arithmetic units

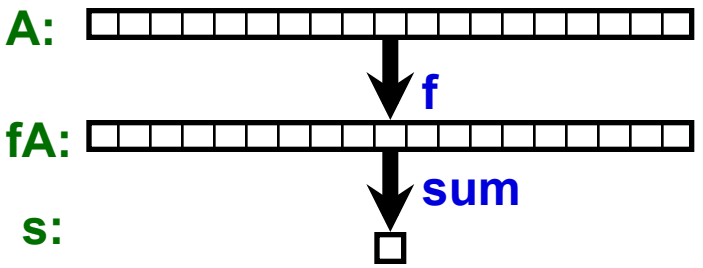
Ideal cost

- These are the natural "abstract" machine models

Data Parallel Programming

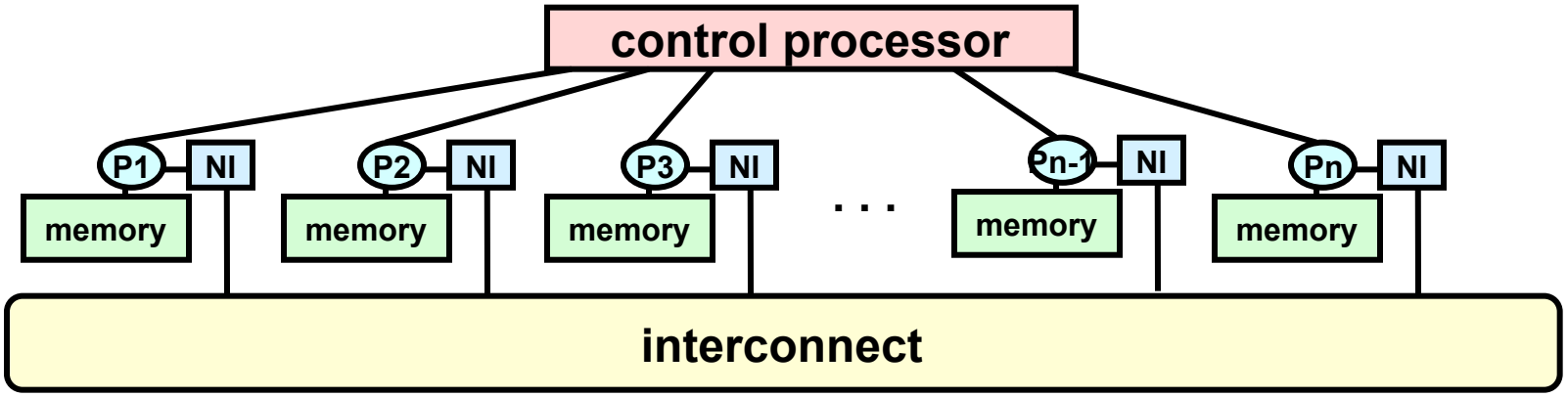
- Single thread of control consisting of parallel operations.
  - A = B+C could mean add two arrays in parallel
- Parallel operations applied to all (or a defined subset) of a data structure, usually an array
  - **Communication is implicit in parallel operators**
  - **Elegant and easy to understand and reason about**
- Drawbacks:
  - **Not all problems fit this model**
  - **Difficult to map onto coarse-grained (i.e., today's) machines**
- CUDA, MPI Collectives, and MapReduce uses these ideas

A = array of all data
fA = f(A)
s = sum(fA)



SIMD System

- A large number of (usually) small processors.
  - **A single "control processor" issues each instruction.**
  - **Each processor executes the same instruction.**
  - **Some processors may be turned off on some instructions.**
- Originally machines were specialized to scientific computing, few made (CM2, Maspar)
- Programming model can be implemented in the compiler
  - **mapping n-fold parallelism to p processors, n >> p, but it's hard (e.g., HPF)**

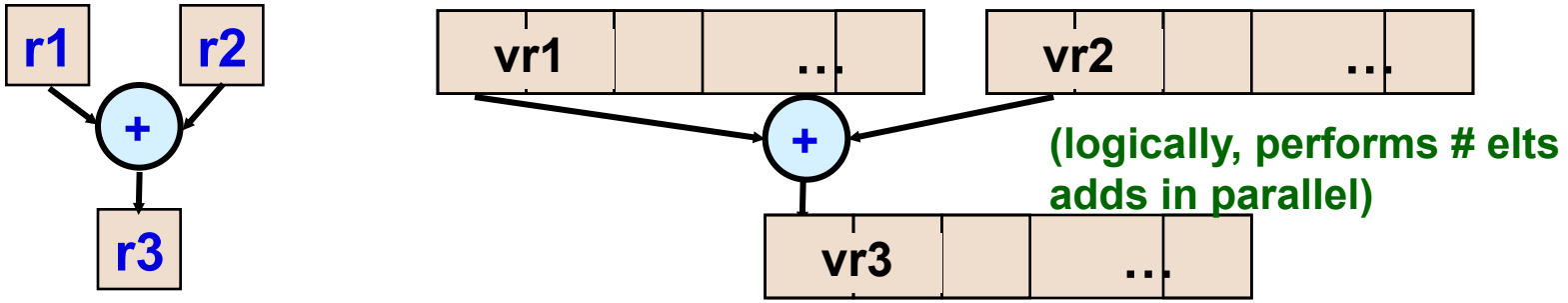


Machine Model Vector Machines

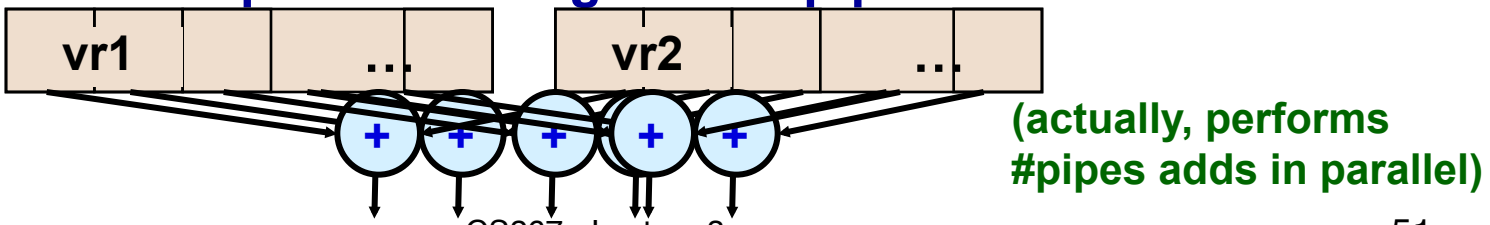
- Vector architectures are based on a single processor
  - **Multiple functional units**
  - **All performing the same operation**
  - **Instructions may specific large amounts of parallelism (e.g., 64-way) but hardware executes only a subset in parallel**
- Historically important
  - **Overtaken by MPPs in the 90s**
- Re-emerging in recent years
  - **At a large scale in the Earth Simulator (NEC SX6) and Cray X1**
  - **At a small scale in SIMD media extensions to microprocessors**
    - **SSE, SSE2 (Intel: Pentium/IA64), AVX, AVX512,…**
    - **Altivec (IBM/Motorola/Apple: PowerPC)**
    - **VIS (Sun: Sparc)**
  - **At a larger scale in GPUs**
- Key idea: Compiler does some of the difficult work of finding parallelism, so the hardware doesn't have to

Vector Processors

- Vector instructions operate on a vector of elements
  - **These are specified as operations on vector registers**



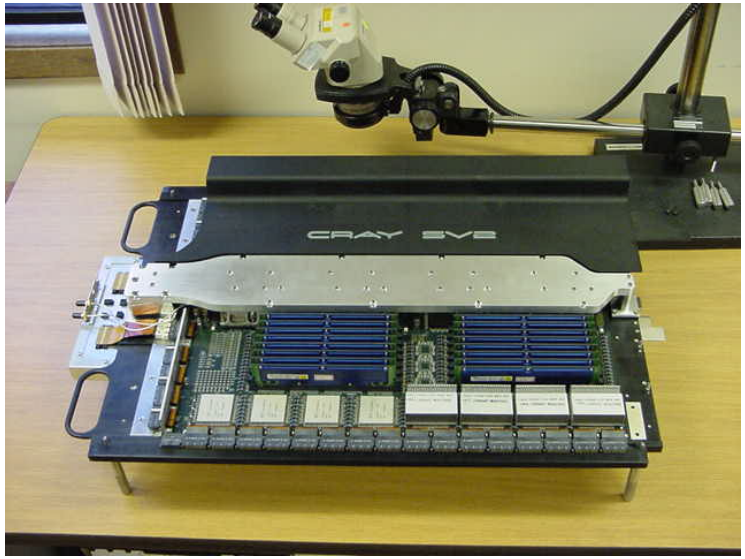
- **A supercomputer vector register holds ~32-64 elts**
  - **The number of elements is larger than the amount of parallel hardware, called vector pipes or lanes, say 2-4**
  - **SIMD register are much smaller (2-8 elts), hardware not virtualized (AVX 256, AVX 512,..)**
- **The hardware performs a full vector operation in**
  - **#elements-per-vector-register / #pipes**



Cray X1: *Parallel Vector Architecture*

Cray combines several technologies in the X1

- **12.8 Gflop/s Vector processors (MSP)**
- **Shared caches (unusual on earlier vector machines)**
- **4 processor nodes sharing up to 64 GB of memory**
- **Single System Image to 4096 Processors**
- **Remote put/get between nodes (faster than MPI)**



Cray X1 Node

- Cray X1 builds a larger "virtual vector", called an MSP
  - 4 SSPs (each a 2-pipe vector processor) make up an MSP
  - Compiler will (try to) vectorize/parallelize across the MSP

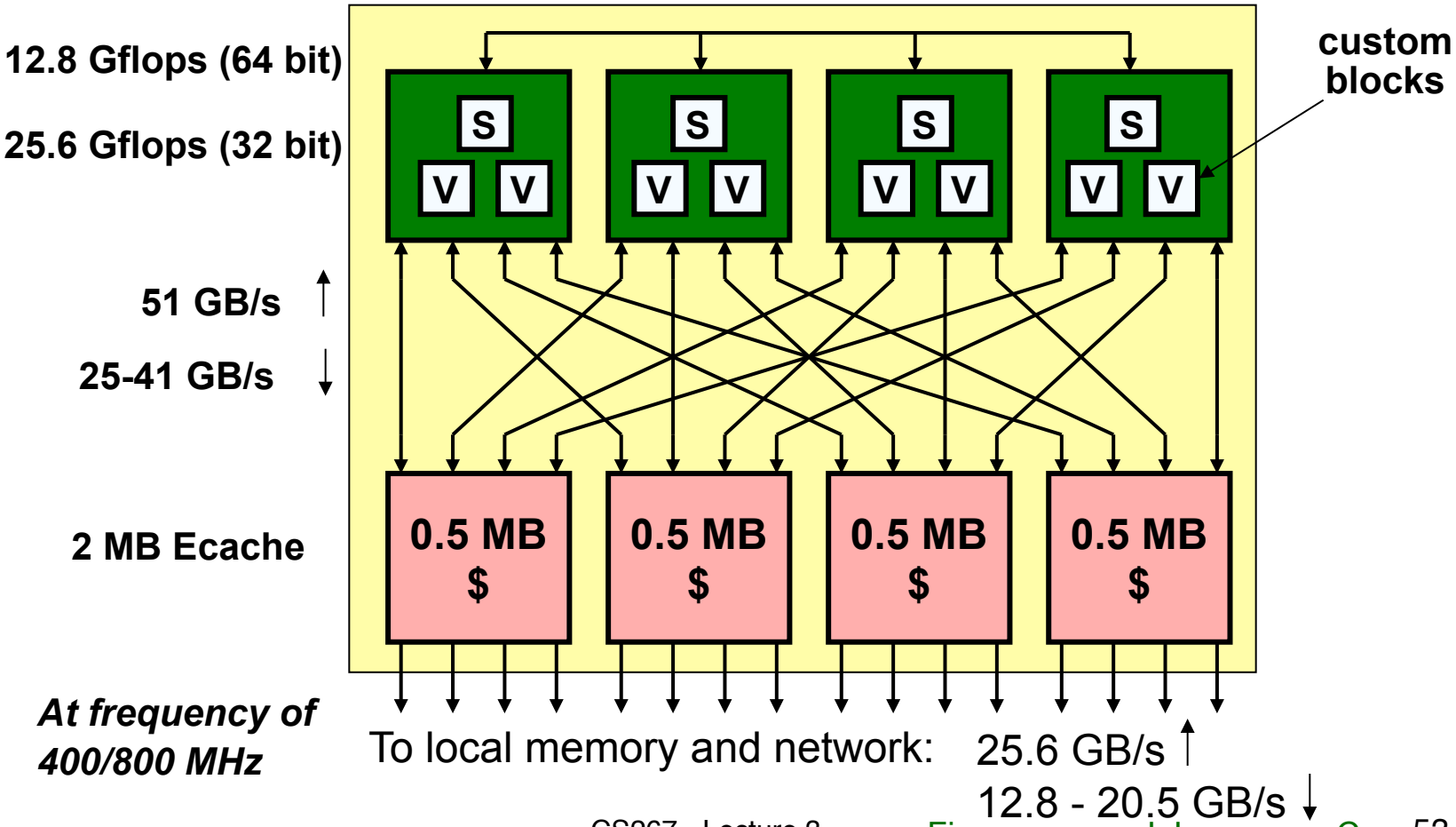


Figure source J. Levesque, Cray

Data Parallel Programming (aka Tricks with Trees)

Some slides from John Gilbert, who borrowed some from Jim Demmel, Kathy Yelick ☺, Alan Edelman, and a cast of thousands …

Our definition for today

- A (pure) data parallel language has
  - A single thread of control, i.e., a serial semantics, which means all behaviors we can see in parallel can also be observed in the serial execution (no races, no nondeterminism)
  - It has operations on aggregate data structures (collections) to (implicitly) express parallelism
- These have a limited expressiveness, but clean and intuitive semantics
- Take advantage of collection-oriented languages, which exist independent of parallelism

Collection Oriented Languages

- **Unary Apply-to-each, e.g., negate elements of vector A**
  - Implicit: -A (APL)
  - Explicit: {-e : e in A } (SETL)
- **Non-unary Apply-to-each**
  - Implicit: A+B
  - Element extension: a+B, add a scalar to a vector
- **Rearranging elements**
  - Permute according to a list of indices (source or target)
- **Nesting: collections inside collections**
- **Example in APL of evaluating a polynomial at x:**

APL Code:

+/(A*{*\1,(((ρA)-1) ρx)))

A = [1 2 3 4] (coefficients)

x = 2

=> $1 + 2x2 + 3x2^2 + 4x2^3 = 41$

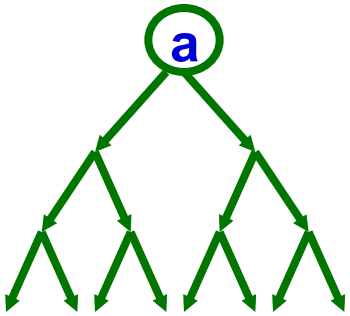
Sipelstein, Jay M. and Blelloch, Guy E., "Collection-oriented languages" (1990)

Parallel Vector Operations

- Vector add: $z = x + y$
  - "embarrassingly parallel" if vectors are aligned
  - may require some communication if not
- DAXPY: $z = a*x + y$ (a is scalar)
  - broadcast a, followed by independent * and +
- DDOT: $s = x^T y = \Sigma_j x[j] * y[j]$
  - Independent * followed by + reduction
- Scan: output vector y based on x: $y_i = \Sigma_{j=1:i} x_j$
- Parallelism is not visible (modulo floating point arithmetic in reduction – what order do +'s happen?)

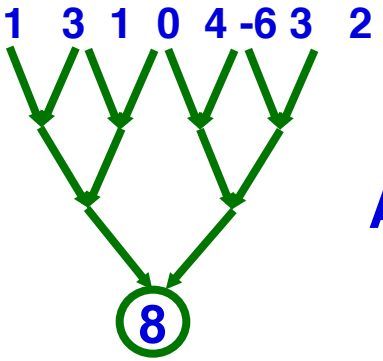
Broadcast and reduction

- Broadcast of 1 value to p processors with log p span



Broadcast

- Reduction of p values to 1 with log p span
- Takes advantage of associativity in +, *, min, max, etc.

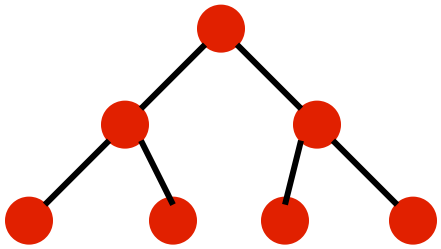


Add-reduction

Log n lower bound on reductions

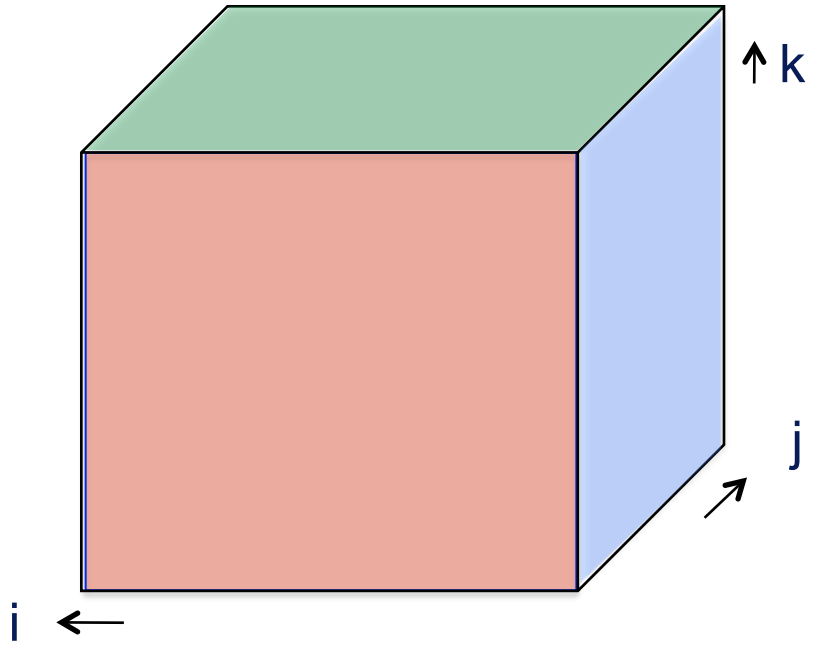
and on any function of n variables!

- Given a function f (x1,…xn) of n input variables and 1 output variable, how fast can we evaluate it in parallel?
- Assume we only have binary operations, one per time step
- After 1 time step, an output can only depend on two inputs
- Use induction to show that after k time units, an output can only depend on 2^k inputs
  - After $\log_2$ n time units, output depends on at most n inputs
- A binary tree performs such a computation



Multiplying n-by-n matrices in O(log n) time

- For all (1 <= i,j,k <= n) P(i,j,k) = A(i,k) * B(k,j)
  - cost = 1 time unit, using n^3 processors
- For all (1 <= i,j <= n) $C(i,j) = \sum_{k=1}^{n} P(i,j,k)$
  - cost = O(log n) time, using n^2 trees with $n^3 / 2$ processors



Put a processor at every point in this cube

What about Scan (aka Parallel Prefix)?

- Definition: the parallel prefix operation takes a binary associative operator $\ominus$, and an array of n elements

  $[a_0, a_1, a_2, \ldots a_{n-1}]$

  and produces the array

  $[a_0, (a_0 \ominus a_1), \ldots (a_0 \ominus a_1 \ominus \ldots \ominus a_{n-1})]$

- Example: add scan of

  [1, 2, 0, 4, 2, 1, 1, 3] is [1, 3, 3, 7, 9, 10, 11, 14]

- Other operators
  - Reals: +, *, min, max
  - Booleans: and, or
  - Matrices: mat mul

Can we parallelize a scan?

- It looks like this:

```
y(0) = 0;
for i = 1:n
    y(i) = y(i-1) + x(i);
```

- The ith iteration of the loop depends completely on the (i-1)st iteration.

- Impossible to parallelize, right?

A clue

x = (1, 2, 3, 4, 5, 6, 7, 8)
y = (1, 3, 6, 10, 15, 21, 28, 36)

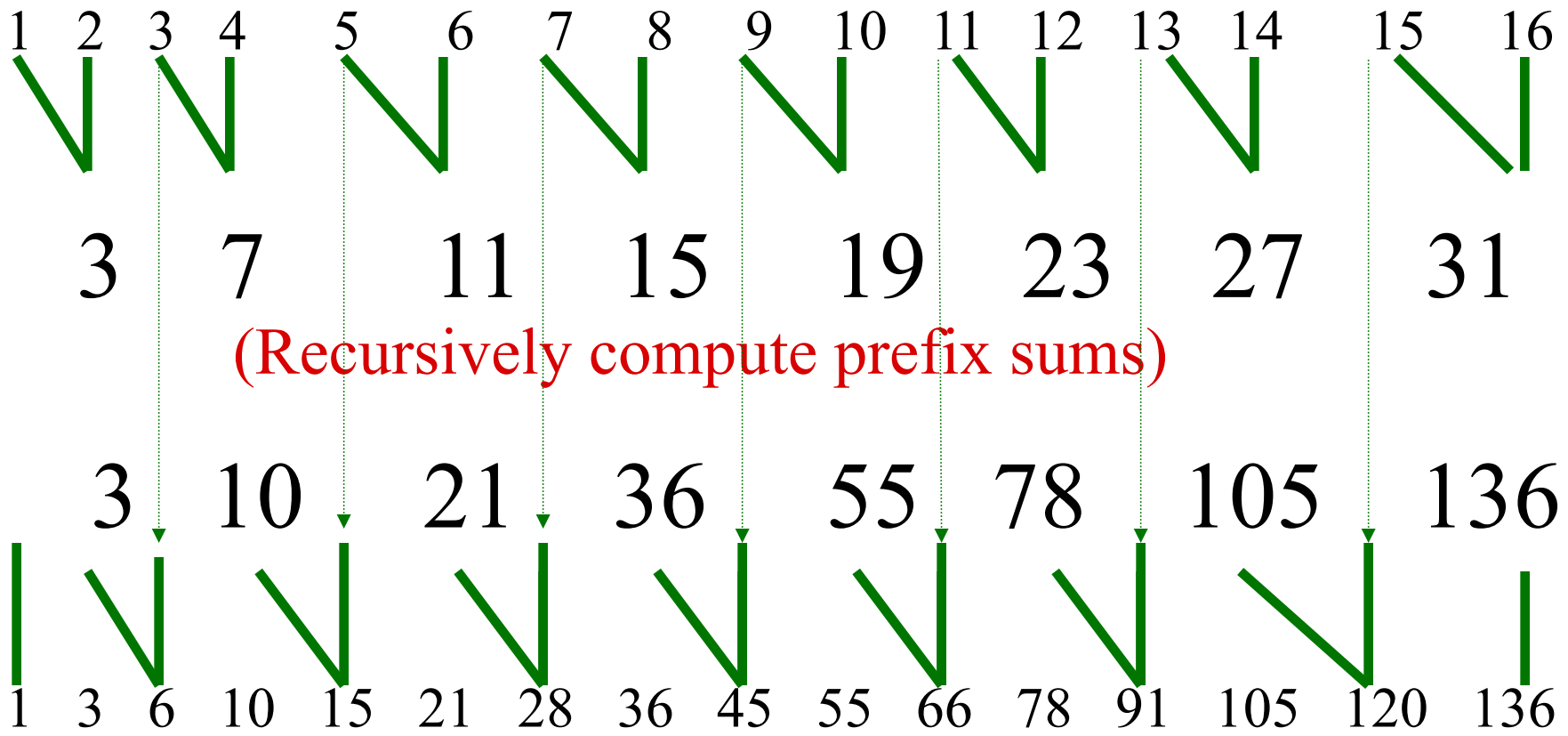
Is there any value in adding, say, 4+5+6+7?

If we separately have 1+2+3, what can we do?

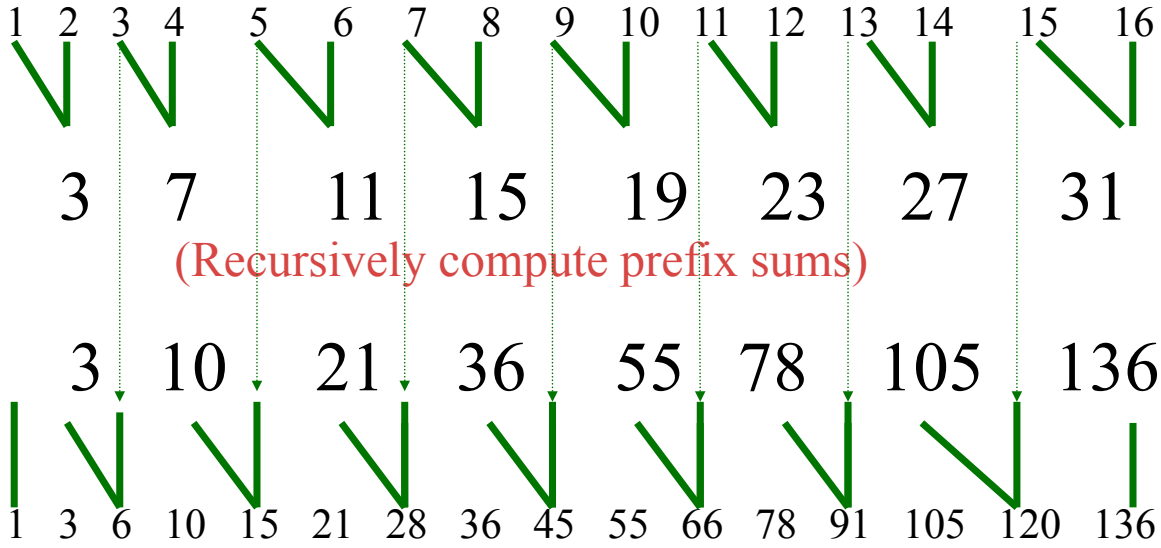
Suppose we added 1+2, 3+4, etc. pairwise -- what could we do?

Prefix sum in parallel

Algorithm: **1. Pairwise sum** **2. Recursive prefix** **3. Pairwise sum**



Parallel prefix cost



Pairwise sum

Recursive prefix

Pairwise sum (update odds)

Time on one processor (work)

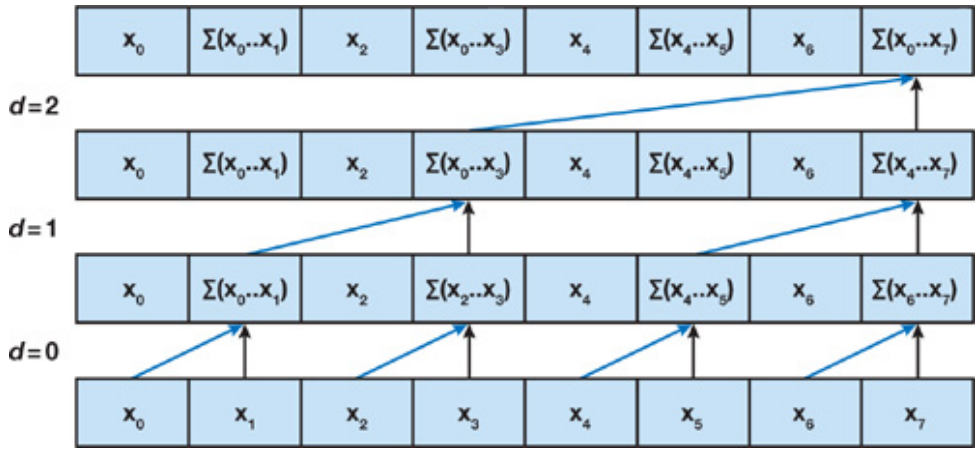
- $T_1(n) = n/2 + n/2 + T_1(n/2) = n + T_1(n/2) = 2n - 1$

Time on unbounded number of processors (span)

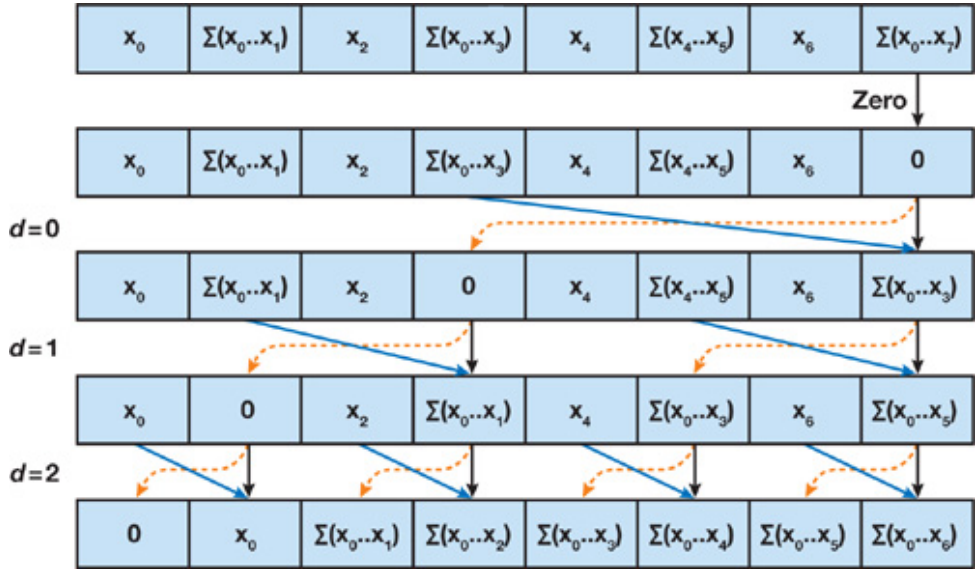
- $T_\infty(n) = 2 \log n$

Parallelism at the cost of more work (2x)!

Non-recursive view of parallel prefix scan



Up-sweep



Down-sweep

This is both work-efficient and space-efficient

Algorithm due to Blelloch
Image due to NVIDIA

Segmented Scans

Inputs = Ordered Pairs

(operand, boolean)

e.g. (x, T) or (x, F)

Change of segment indicated by switching T/F

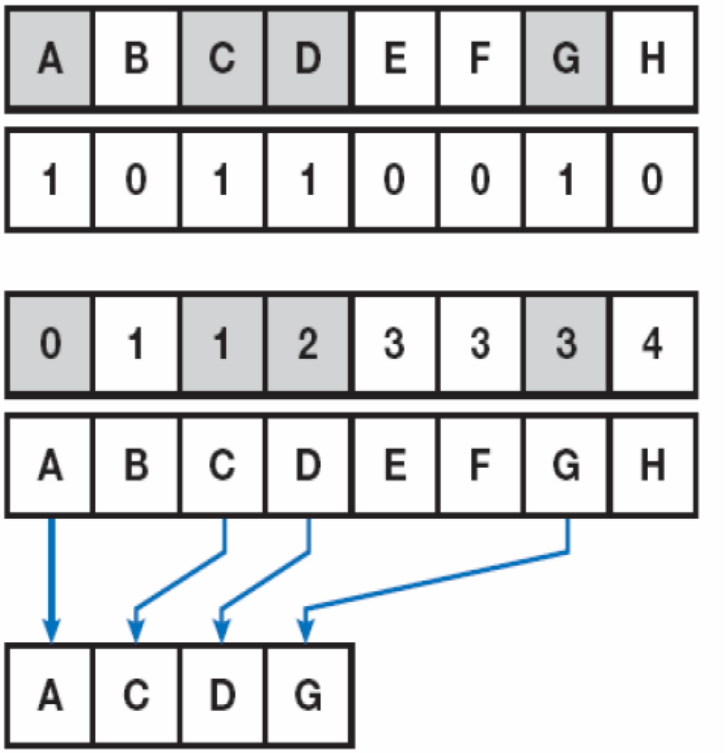
$+_2$	(y, T)	(y, F)
(x, T)	(x+y, T)	(y, F)
(x, F)	(y, T)	(x⊕y, F)

e. g.	1	2	3	4	5	6	7	8
	T	T	F	F	F	T	F	T
Result	1	3	3	7	12	6	7	8

Scans are useful for many things (partial list here)

- Reduction and broadcast in O(log n) time
- Parallel prefix (scan) in O(log n) time
- Adding two n-bit integers in O(log n) time
- Multiplying n-by-n matrices in O(log n) time
- Inverting n-by-n triangular matrices in $O(\log^2 n)$ time
- Inverting n-by-n dense matrices in $O(\log^2 n)$ time
- Evaluating arbitrary expressions in O(log n) time
- Evaluating recurrences in O(log n) time
- "2D parallel prefix", for image segmentation (Catanzaro & Keutzer)
- Sparse-Matrix-Vector-Multiply (SpMV) using Segmented Scan
- Parallel page layout in a browser (Leo Meyerovich, Ras Bodik)
- Solving n-by-n tridiagonal matrices in O(log n) time
- Traversing linked lists
- Computing minimal spanning trees
- Computing convex hulls of point sets…

Application: Stream Compaction



Input array of “good/bad” values
Output: in-order array of only good ones

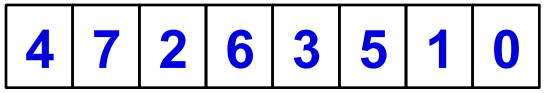
Create an array of flags for good ones

Perform an add scan on the flags

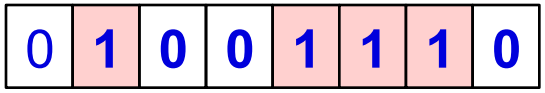
“Scatter” input to output for every processor with a flag of 1

Image and example from J. Reif

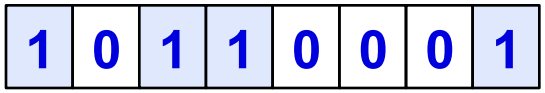
Application: Radix Sort



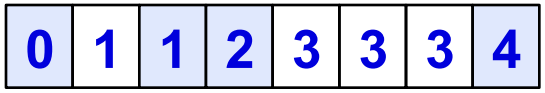
Input



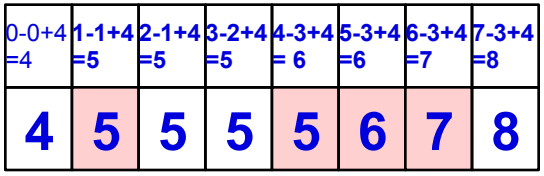
b = last bit



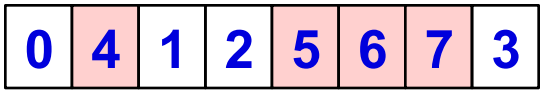
e = Mask of all evens (last bit = 0)



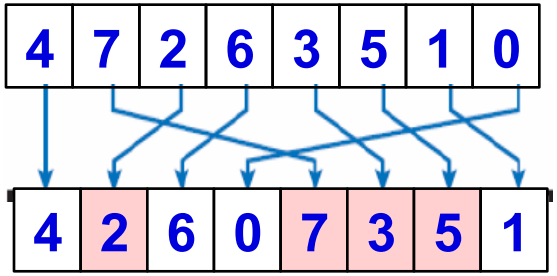
f = Scans of evens
totalFalses = last element



t = i – f + totalFalses



d = if b then t else f



Scatter input using d as index

Repeat with next bit to left until done

E.g., Fibonacci via Matrix Multiply Prefix

$$F_{n+1} = F_n + F_{n-1}$$

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

Can compute all F_n by matmul_prefix on

$$\left[\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \right]$$

then select the upper left entry

Adding two n-bit integers in O(log n) time

- Let a = a[n-1]a[n-2]…a[0] and b = b[n-1]b[n-2]…b[0] be two n-bit binary numbers
- We want their sum s = a+b = s[n]s[n-1]…s[0]

```
c[-1] = 0          … rightmost carry bit
for i = 0 to n-1
    c[i] = ( (a[i] xor b[i])  and  c[i-1] )  or  ( a[i]  and  b[i] )   ... next carry bit
    s[i] = ( a[i] xor b[i] ) xor c[i-1]
```

- Challenge: compute all c[i] in O(log n) time via parallel prefix

```
for all (0 <= i <= n-1)  p[i] = a[i] xor b[i]      … propagate bit
for all (0 <= i <= n-1)  g[i] = a[i] and b[i]      … generate bit
```

$$\begin{pmatrix} c[i] \\ 1 \end{pmatrix} = \begin{pmatrix} (\, p[i] \text{ and } c[i-1] \,) \text{ or } g[i] \\ 1 \end{pmatrix} = \begin{pmatrix} p[i] & g[i] \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} c[i-1] \\ 1 \end{pmatrix} = C[i] * \begin{pmatrix} c[i-1] \\ 1 \end{pmatrix}$$

… 2-by-2 Boolean matrix multiplication (associative)

$$= C[i] * C[i-1] * \ldots C[0] * \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

… evaluate each P[i] = C[i] * C[i-1] * … * C[0] by parallel prefix

- Used in all computers to -- Carry look-ahead addition

E.g., Using Scans for Array Compression

- Given an array of n elements

  $[a_0, a_1, a_2, \ldots a_{n-1}]$

  and an array of flags

  [1,0,1,1,0,0,1,…]

  compress the flagged elements into

  $[a_0, a_2, a_3, a_6, \ldots]$

- Compute an add scan of [0, flags] :

  [0,1,1,2,3,3,4,…]

- Gives the index of the i^{th} element in the compressed array
  - If the flag for this element is 1, write it into the result array at the given position

Lexical analysis (tokenizing, scanning)

- Given a language of:
  - Identifiers: string of chars
  - Strings: in double quotes
  - Ops: +,-,*,=,<,>,<=, >=

TABLE I. A Finite-State Automaton for Recognizing Tokens

Old State	Character Read													
•	A	B	...	Y	Z	+	−	*	<	>	=	"	Space	New line
N	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
A	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
Z	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
*	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
<	A	A	...	A	A	*	*	*	<	<	=	Q	N	N
=	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
Q	S	S	...	S	S	S	S	S	S	S	S	E	S	S
S	S	S	...	S	S	S	S	S	S	S	S	E	S	S
E	E	E	...	E	E	*	*	*	<	<	*	S	N	N

- Lexical analysis
  - Replace every character in the string with the array representation of its state-to-state function (column).
  - Perform a parallel-prefix operation with ⊕ as the array composition. Each character becomes an array representing the state-to-state function for that prefix.
  - Use initial state (row 1) to index into these arrays.

Hillis and Steele, CACM 1986

Inverting triangular n-by-n matrices in $O(\log^2 n)$ time

- Fact: $\begin{pmatrix} A & 0 \\ C & B \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ -B^{-1}CA^{-1} & B^{-1} \end{pmatrix}$
- Function Tri_Inv(T) … assume n = dim(T) = 2^m for simplicity

```
If T is 1-by-1
   return 1/T
else
   … Write T = [A  0]
               [C  B]
   In parallel do {
         invA = Tri_Inv(A)
         invB = Tri_Inv(B)  }      … implicitly uses a tree
   newC = -invB * C * invA
   Return [invA    0  ]
          [newC  invB ]
```

- time(Tri_Inv(n)) = time(Tri_Inv(n/2)) + O(log(n))
  - Change variable to m = log n to get time(Tri_Inv(n)) = $O(\log^2 n)$

Inverting Dense n-by-n matrices in $O(\log^2 n)$ time

- Lemma 1: Cayley-Hamilton Theorem
  - expression for A^{-1} via characteristic polynomial in A
- Lemma 2: Newton's Identities
  - Triangular system of equations for coefficients of characteristic polynomial, where matrix entries = s_k
- Lemma 3: $s_k = \text{trace}(A^k) = \sum_{i=1}^{n} A^k[i,i]$
- Csanky's Algorithm (1976)

1) Compute the powers $A^2, A^3, \ldots, A^{n-1}$ by parallel prefix
cost = $O(\log^2 n)$
2) Compute the traces $s_k = \text{trace}(A^k)$
cost = $O(\log n)$
3) Solve Newton identities for coefficients of characteristic polynomial
cost = $O(\log^2 n)$
4) Evaluate A^{-1} using Cayley-Hamilton Theorem
cost = $O(\log n)$

- **Completely numerically unstable**

Evaluating arbitrary expressions

- Let E be an arbitrary expression formed from +, -, *, /, parentheses, and n variables, where each appearance of each variable is counted separately
- Can think of E as arbitrary expression tree with n leaves (the variables) and internal nodes labelled by +, -, * and /
- Theorem (Brent): E can be evaluated with O(log n) span, if we reorganize it using laws of commutativity, associativity and distributivity
- Sketch of (modern) proof: evaluate expression tree E greedily by
  - collapsing all leaves into their parents at each time step
  - evaluating all "chains" in E with parallel prefix

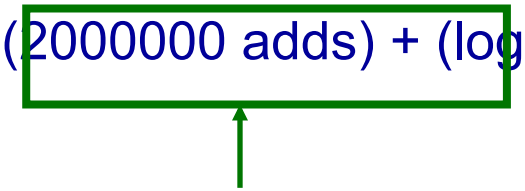
Other applications of scan = parallel prefix

- There are many applications of scans, some more obvious than others
  - add multi-precision numbers (represented as array of numbers)
  - evaluate recurrences, expressions
  - solve tridiagonal systems (but numerically unstable!)
  - implement bucket sort and radix sort
  - to dynamically allocate processors
  - to search for regular expression (e.g., grep)
  - many others…
- Names: +\ (APL), cumsum (Matlab), MPI_SCAN
- Note: 2n operations used when only n-1 needed

The myth of log n

- **The $\log_2 n$ span is not the main reason for the usefulness of parallel prefix.**

- **Say n = 1000000p** (1000000 elements per processor)
  - Cost = (2000000 adds) + ($\log_2 P$ message passings)



fast & embarassingly parallel

(2000000 local adds are serial for each processor, of course)

Key to implementing data parallel algorithms on clusters, MPPs, i.e., modern supercomputers

Lessons from Data Parallel Languages

- Sequential semantics (or nearly) is very nice
  - Debugging is much easier without non-determinism
  - Correctness easier to reason about
- Cost model is independent of number of processors
  - How much inherent parallelism
- Need to “throttle” parallelism
  - n >> p can be hard to map, especially with nesting
  - Memory use is a problem

See: Blelloch “NESL Revisited”, Intel Workshop 2006