Single processor tuning (1/2)

Prof. Richard Vuduc
Georgia Institute of Technology

CSE/CS 8803 PNA: Parallel Numerical Algorithms

[L.14] Thursday, February 21, 2008

Today's sources

- CS 267 (Yelick @ UCB; Spring 2007)
- "*A survey of out-of-core algorithms in numerical linear algebra*," by Toledo (1999)
- "*A family of high-performance matrix multiplication algorithms*," by Gunnels, *et al*. (2006)
- "*On reducing TLB misses in matrix multiplication*," by Goto and van de Geijn (2002)
- "*Is search really necessary to generate high-performance BLAS?*" by Yotov, *et al*. (2005)

Review: Accuracy, stability, and parallelism

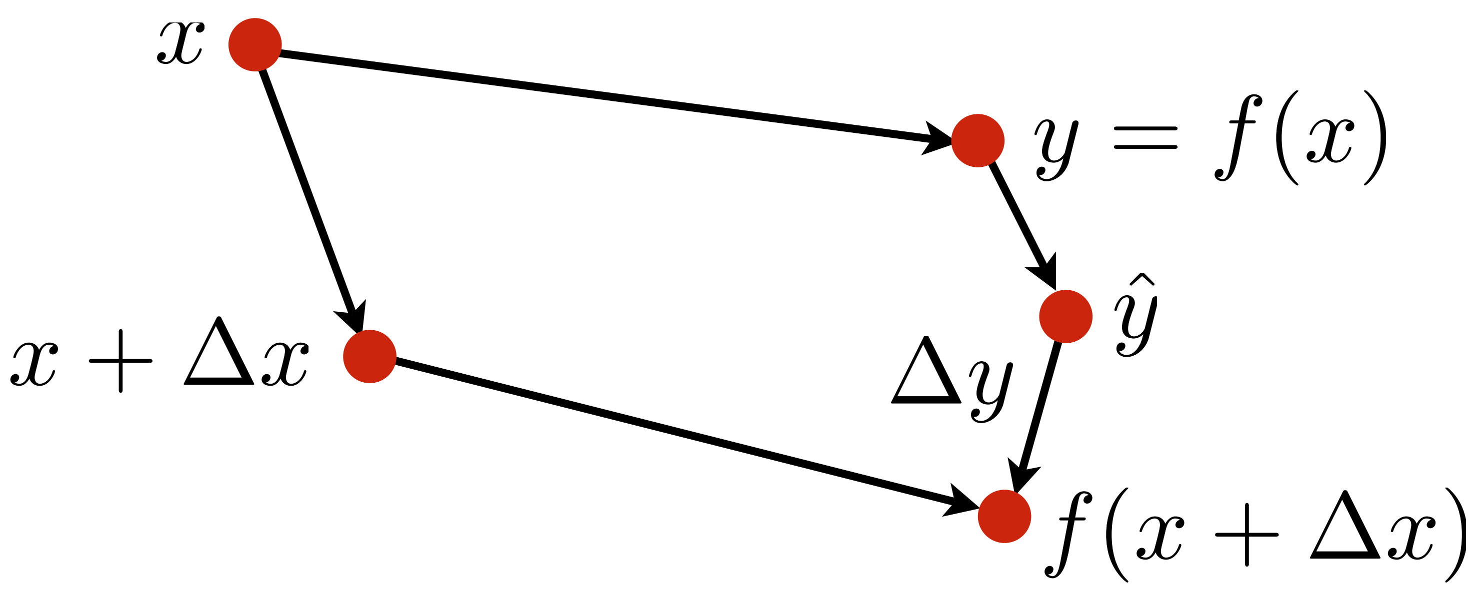
The impact of parallelism on numerical algorithms

- **Larger problems** magnify errors: Round-off, ill-conditioning, instabilities
- **Reproducibility**: a + (b + c) ≠ (a + b) + c
- Fast **parallel algorithm** may be much **less stable** than fast serial algorithm
- **Flops cheaper** than communication
- **Speeds at different precisions** may vary significantly [*e.g.*, SSE_k, Cell]
- Perils of **arithmetic heterogenity**, *e.g.*, CPU vs. GPU support of IEEE

Mixed (forward-backward) stability

- Computed answer "near" exact solution of a nearby problem

$$\Delta x, \Delta y: \qquad \hat{y} + \Delta y = f(x + \Delta x)$$



Conditioning: Relating forward and backward error

$$\left|\frac{\hat{y} - y}{y}\right| \lesssim \left|\frac{x f'(x)}{f(x)}\right| \cdot \left|\frac{\Delta x}{x}\right|$$

- Define **(relative) condition number**:

$$c(x) = \left|\frac{x f'(x)}{f(x)}\right|$$

- Roughly: **(Forward error) ≤ (Condition number) * (Backward error)**

Mixed-precision iterative refinement

- Inner-loop of mixed-precision iterative refinement algorithm:

  Single, O(n^3) ⇒ $\hat{x} = \text{Estimated solution to } Ax = b$

  Double, O(n^2) ⇒ $\hat{r} \leftarrow b - A \cdot \hat{x}$

  Single, O(n^2) ⇒ $\text{Solve } A \cdot \hat{d} = \hat{r}$

  Double, O(n) ⇒ $\hat{x}^{(\text{improved})} \leftarrow \hat{x} + \hat{d}$

- **Theorem**: Repeated iterative refinement converges by η at each stage, and

$$x^{(t)} \equiv \text{Estimate at iteration } t, \text{ in precision } \epsilon$$

$$r^{(t)} \equiv \text{Residual, in precision } \epsilon^2$$

$$\eta \equiv \epsilon \cdot ||\ |A^{-1}| \cdot |\hat{L}| \cdot |\hat{U}|\ ||_\infty < 1$$

$$\frac{||x^{(t)} - x||_\infty}{||x||_\infty} \rightarrow O(\epsilon)$$

⇐ Independent of κ(*A*)!

Obstacles to fast and stable parallel numerical algorithms

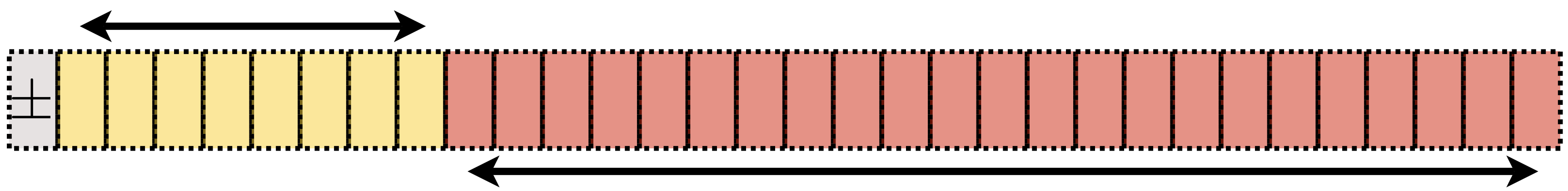
- **Algorithms that work on small problems may fail at large sizes**
  - Round-off accumulates
  - Condition number increases
  - Probability of “random instability” increases

- **Fast (parallel) algorithm may be less stable $\Rightarrow$ trade-off**

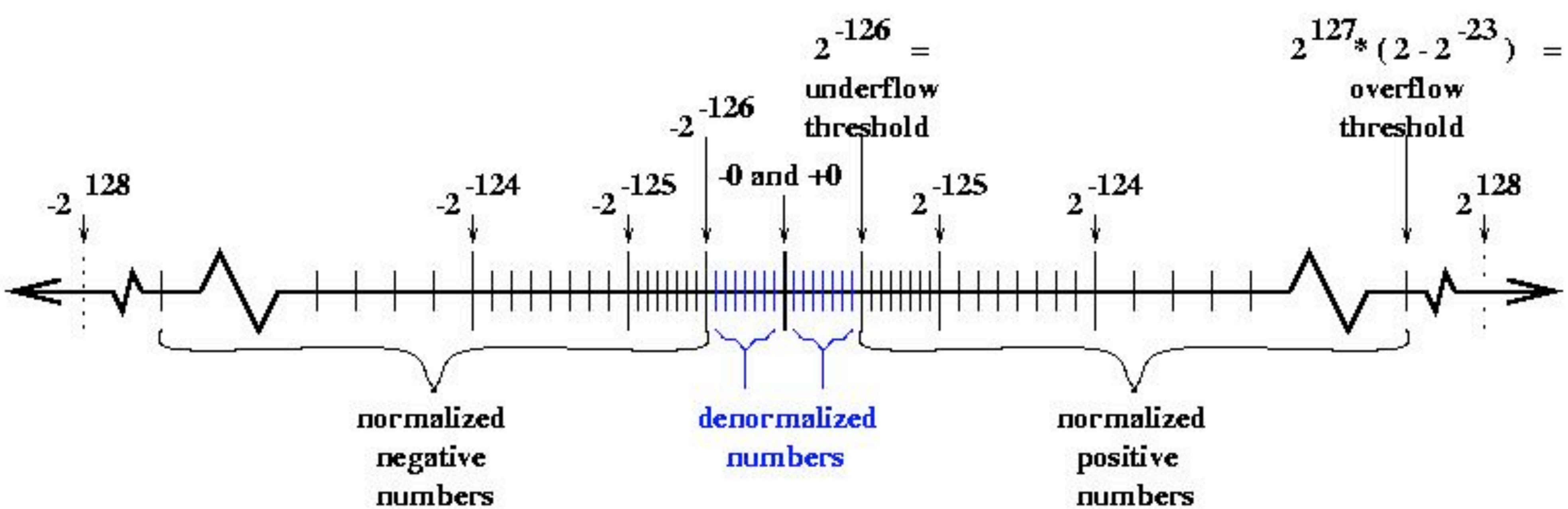
$$y = \pm m \times 2^{e-t}$$

$$y \neq 0 \implies m \geq 2^{t-1}$$ ← **"Normalized"**

$$-125 = e_{\min} \leq e \leq e_{\max} = 128$$



$$0 \leq m < 2^{24} \approx 16 \text{ million}$$

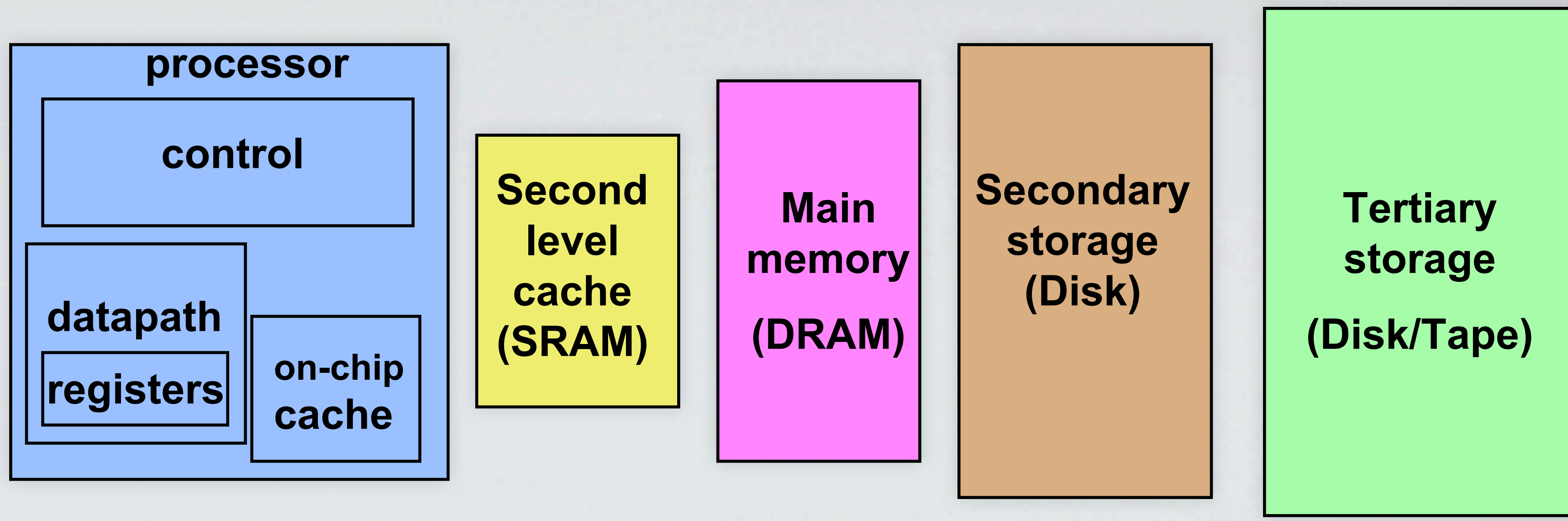


IEEE formats

$e_{\min} \leq e \leq e_{\max}$ $0 \leq m < 2^t$

Format	Total bits	Exp. bits (e_{min}, e_{max})	t-1	ε	Fortran / C
Single	32	8 (-125, 128)	23	6×10^{-8}	`REAL*4` `float`
Double	64	11 (-1021, 1024)	52	10^{-16}	`REAL*8` `double`
Extended (Intel)	80	15 (-16381, 16384)	64	5×10^{-20}	`REAL*10` `long double`

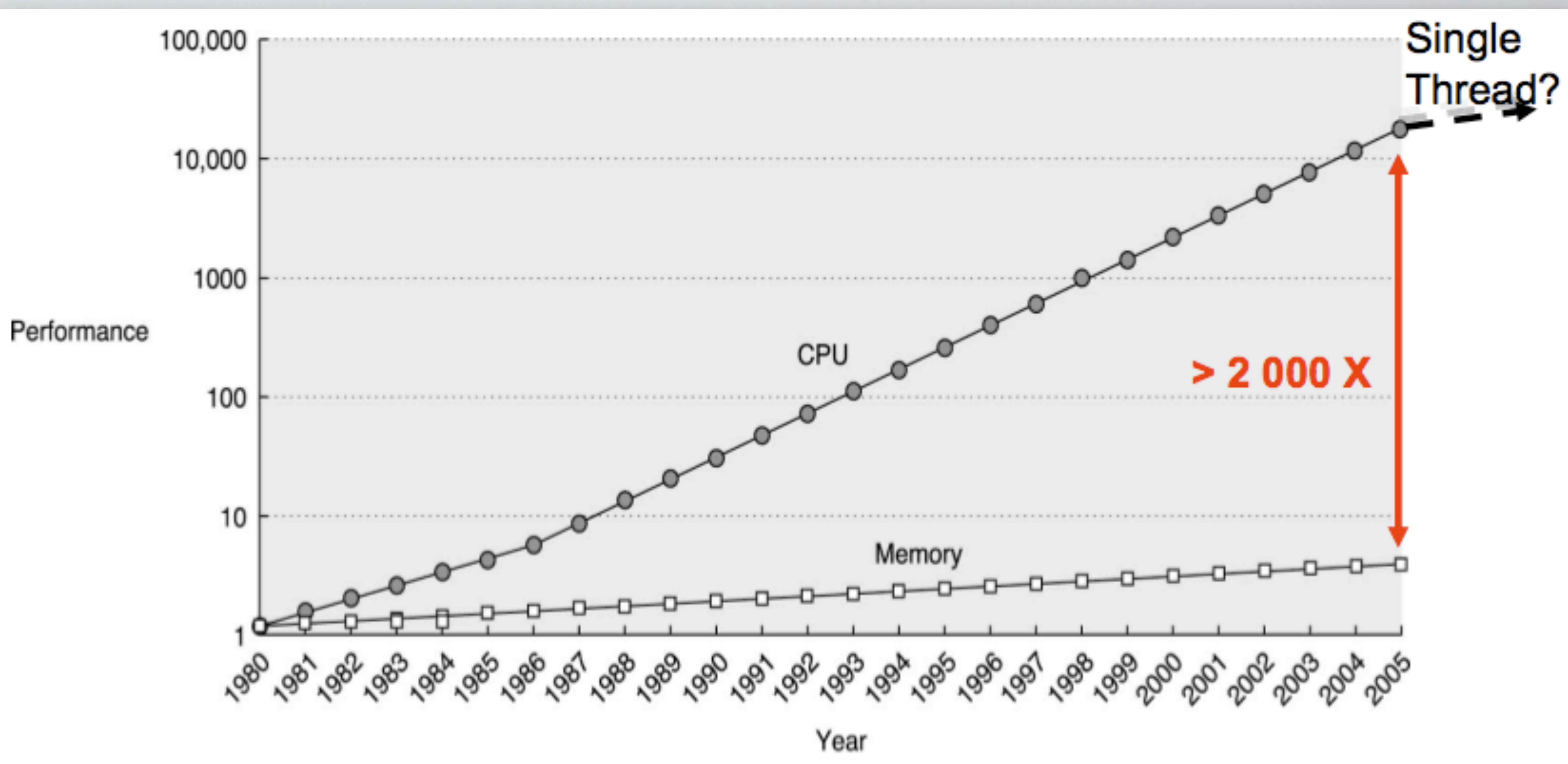
Reasoning about memory hierarchies



Cost	1ns	10ns	100ns	10ms	10sec
Size	B	KB	MB	GB	TB

Recall: Memory hierarchies.

Cost of accessing data depends on where data lives.



Memory hierarchies reflect growing processor-memory speed gap.

Dealing with high memory latency

- Use caches as fast memory
  - Store data that will be reused many times: **temporal locality**
  - Save chunks of contiguous data: **spatial locality**
- Exploit fact that bandwidth improves faster than latency: **prefetch**

- Modern processors automate cache management
  - All loads cached automatically (LRU), and loaded in chunks (*cache line size*)
  - Typical to have a hardware prefetcher that detects simple patterns

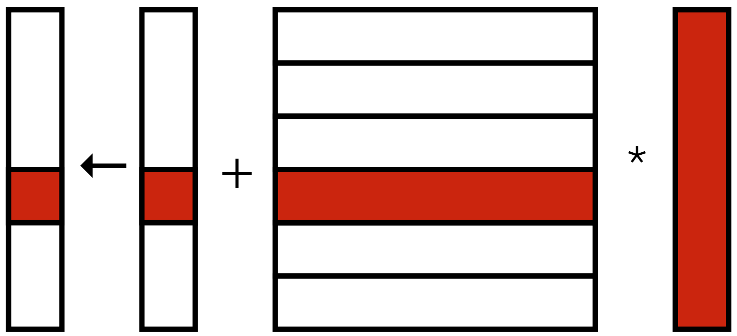
A simple model of memory

$$
\begin{aligned}
m &\equiv \text{No. words moved from slow to fast memory} \\
f &\equiv \text{No. of flops} \\
\alpha &\equiv \text{Time per slow memory op.} \\
\tau &\equiv \text{Time per flop} \\
q &\equiv \frac{f}{m} = \text{Flop-to-mop ratio} \quad \Leftarrow \text{Computational intensity} \\
T &= f \cdot \tau + m \cdot \alpha = f \cdot \tau \cdot \left(1 + \frac{\alpha}{\tau} \cdot \frac{1}{q}\right)
\end{aligned}
$$

Machine balance

Example: Matrix-vector multiply

// Implements $y \leftarrow y + A \cdot x$



for $i \leftarrow 1$ **to** n **do**

for $j \leftarrow 1$ **to** n **do**

$$y_i \leftarrow y_i + a_{ij} \cdot x_j$$

Example: Matrix-vector multiply

// Implements $y \leftarrow y + A \cdot x$

// Read x (into fast memory)

// Read y

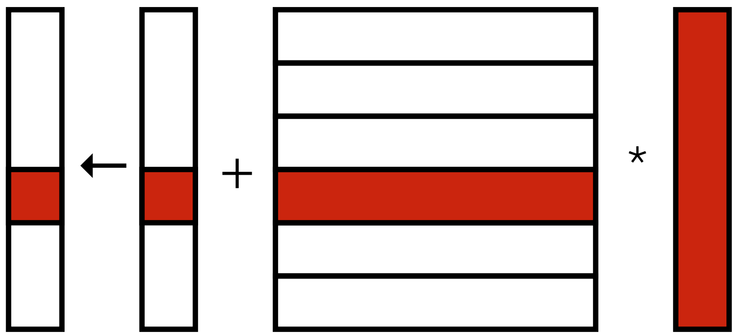
for $i \leftarrow 1$ **to** n **do**

  // Read $a_{i,\star}$

  for $j \leftarrow 1$ **to** n **do**

    $y_i \leftarrow y_i + a_{ij} \cdot x_j$

// Write y to slow memory



Example: Matrix-vector multiply

// Implements $y \leftarrow y + A \cdot x$

// Read x (into fast memory)

// Read y

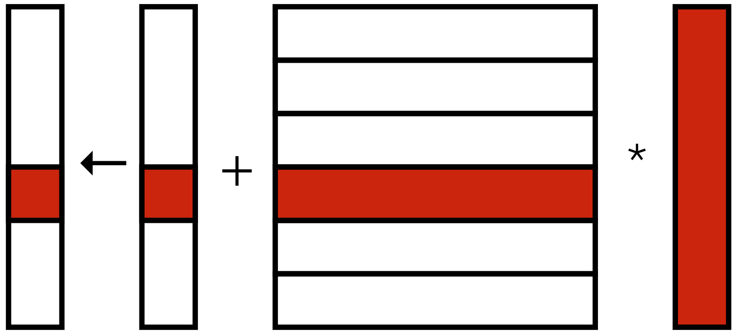
for $i \leftarrow 1$ **to** n **do**

    // Read $a_{i,\star}$

    for $j \leftarrow 1$ **to** n **do**

        $y_i \leftarrow y_i + a_{ij} \cdot x_j$

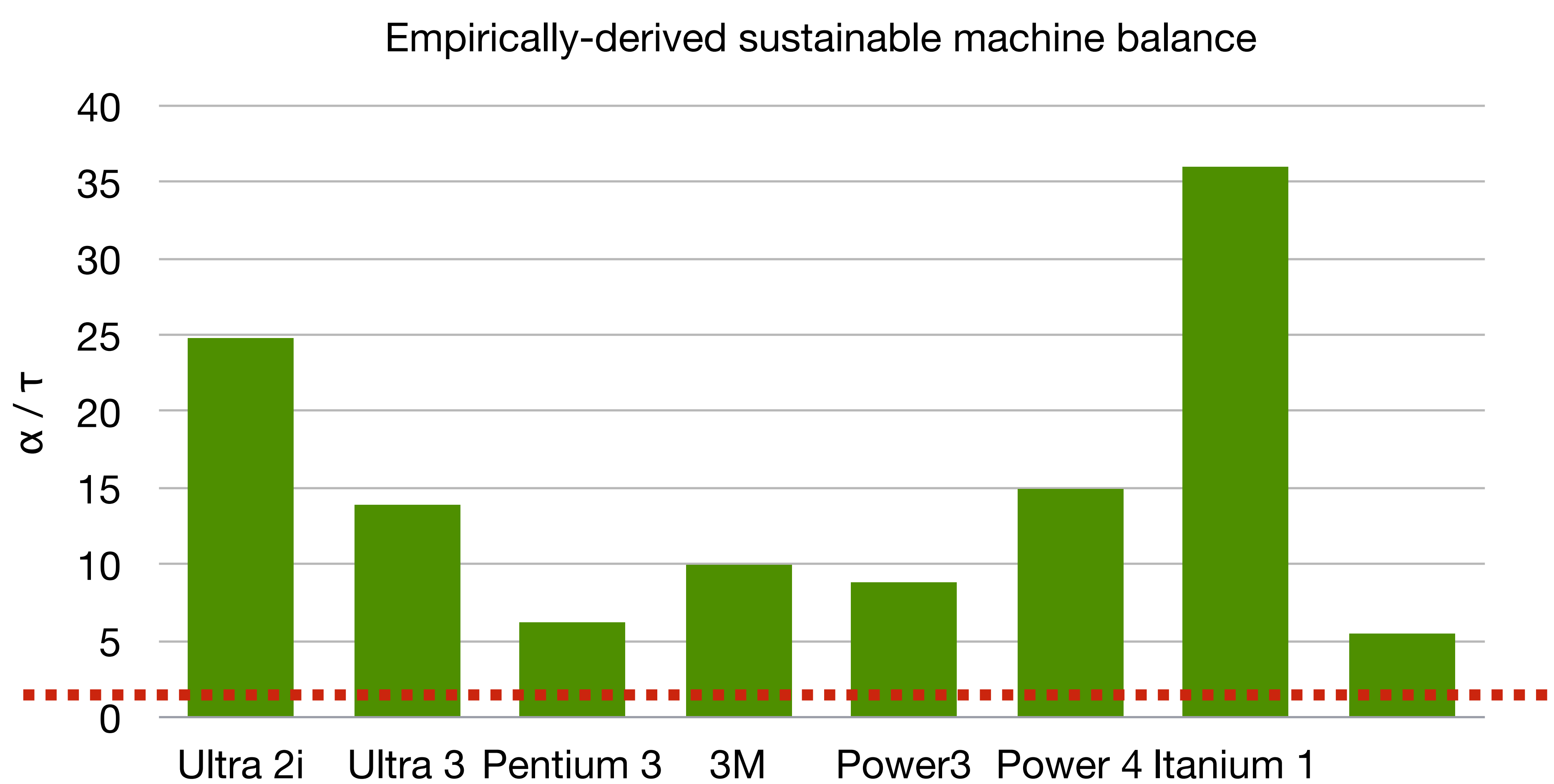
// Write y to slow memory



$$
\begin{aligned}
f &= 2n^2 \\
m &= 3n + n^2 \\
q &\approx 2 \\
&\Downarrow \\
\frac{T}{f \cdot \tau} &\approx 1 + \frac{\alpha}{\tau} \cdot \frac{1}{2}
\end{aligned}
$$

Machine balance, α / τ

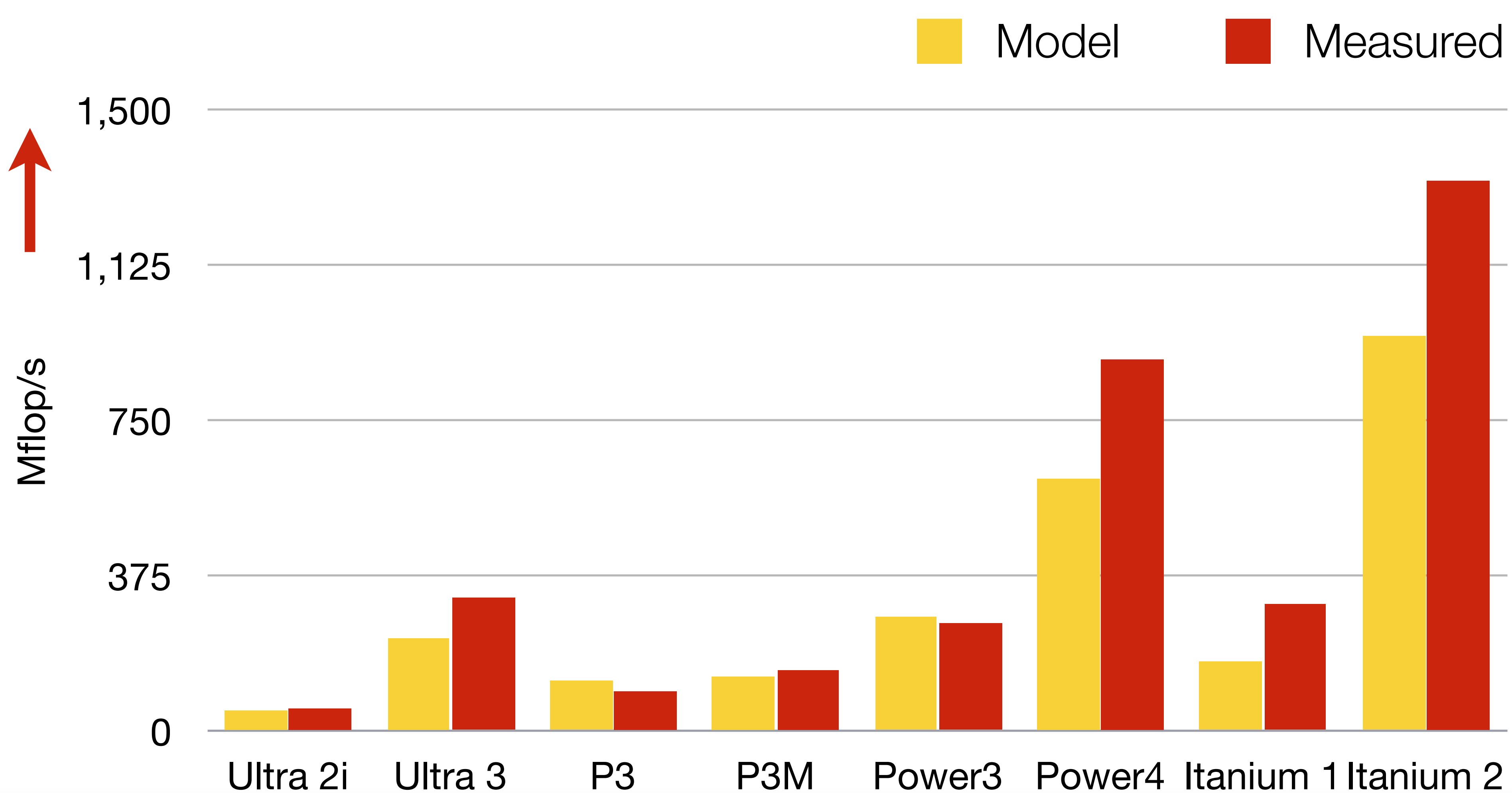
[See my thesis]



Simplifying assumptions

- Ignored flop/mop parallelism within processor → drop arithmetic term
- Assumed fast memory large enough to hold vectors
- Assumed no-cost fast memory access
- Memory latency is constant, charged per word
  - Ignored cache lines / block transfers
  - Ignored bandwidth

Predictive accuracy of this model



Naive matrix-matrix multiply

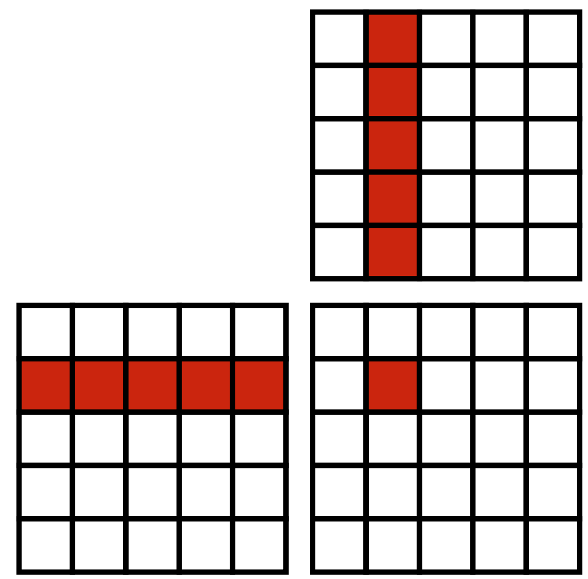
// Implements $C \leftarrow C + A \cdot B$

for $i \leftarrow 1$ **to** n **do**

  for $j \leftarrow 1$ **to** n **do**

    for $k \leftarrow 1$ **to** n **do**

      $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$



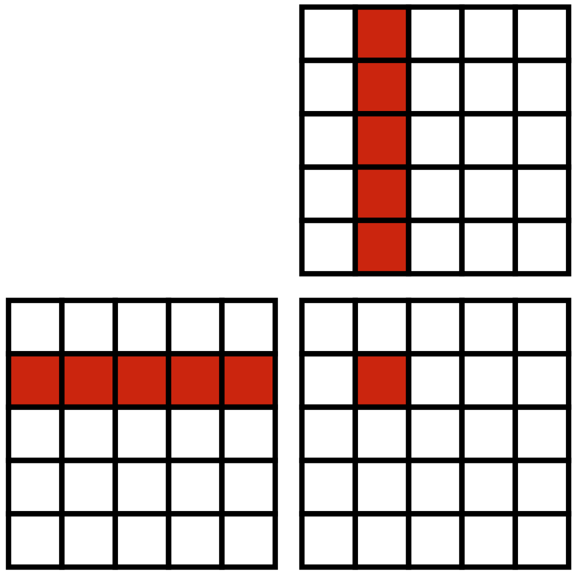
$$f = 2n^3$$

Best case ⇒ $m \geq 4n^2$

$$\frac{T}{f \cdot \tau} \geq 1 + \frac{\alpha}{\tau} \cdot \frac{2}{n}$$

Naive matrix-matrix multiply

```
// Implements C ← C + A · B
for i ← 1 to n do
// Read row a_{i,⋆}
  for j ← 1 to n do
  // Read col b_{⋆,j}
  // Read c_{i,j}
    for k ← 1 to n do
      c_ij ← c_ij + a_ik · b_kj
  // Write c_ij to slow memory
```



$$f = 2n^3$$

$$m = n^3 + 3n^2$$

$$\frac{T}{f \cdot \tau} \approx 1 + \frac{\alpha}{\tau} \cdot \frac{1}{2}$$

Blocked (tiled) matrix multiply

// Let I, J, K = blocks of b indices

for $I \leftarrow$ index blocks 1 **to** $\frac{n}{b}$ **do**

for $J \leftarrow$ index blocks 1 **to** $\frac{n}{b}$ **do**

// Read block C_{IJ}

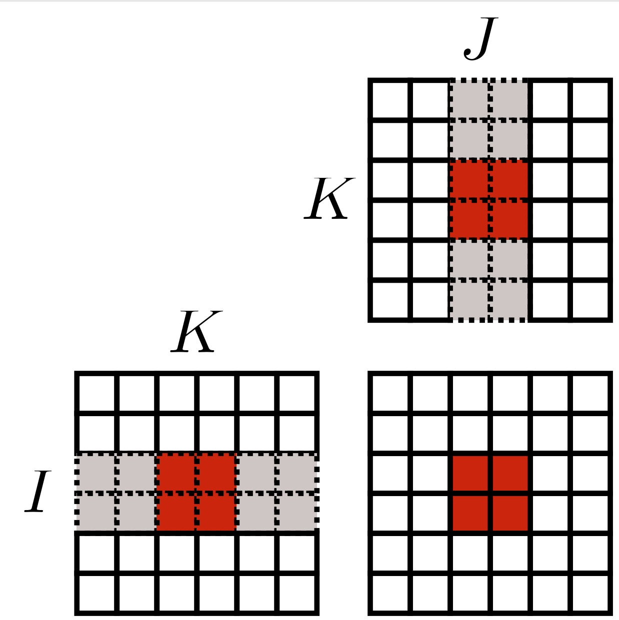
for $K \leftarrow$ index blocks 1 **to** $\frac{n}{b}$ **do**

// Read block A_{IK}

// Read block B_{KJ}

$B_{IJ} \leftarrow c_{IJ} + A_{IK} \cdot B_{KJ}$

// Write C_{IJ} to slow memory



Blocked (tiled) matrix multiply

$$m \approx \frac{n^3}{b} \quad \Longrightarrow \quad q \approx b$$

$$\frac{T}{f \cdot \tau} \quad = \quad 1 + \frac{\alpha}{\tau} \cdot \frac{1}{b}$$

Architectural implications

Arch.	≈ α / τ	M
Ultra 2i	25	1.5 MB
Ultra 3	14	460 KB
Pentium 3	6.3	94 KB
P-3M	10	240 KB
Power3	8.8	180 KB
Power4	15	527 KB
Itanium 1	36	3.0 MB
Itanium 2	5.5	71 KB

$$\begin{aligned} M &\equiv \text{Size of fast mem.} \\ 3b^2 &\leq M \\ q &\approx b \\ &\Downarrow \\ M &\geq 3q^2 \end{aligned}$$

$$\begin{aligned} 1 + \frac{\alpha}{\tau} \cdot \frac{1}{q} &< 1.1 \\ \Longrightarrow M &\geq 300 \left(\frac{\alpha}{\tau}\right)^2 \end{aligned}$$

"M" in bytes to 2 digits; assumes 8-byte (double-precision) words

Can we do better?

$$
\begin{aligned}
b &= O\left(\sqrt{M}\right) \\
\Longrightarrow m &= O\left(\frac{n^3}{b}\right) = O\left(\frac{n^3}{\sqrt{M}}\right)
\end{aligned}
$$

Bounding amount of I/O possible

- Consider a schedule in phases of exactly *M* transfers each (except last)
- *Definition*: *c(i,j)* is **live** during phase *p* if ...
  - … for some *k*, we compute *a(i,k)* * *b(k, j)*;
  - *and* some partial sum of *c(i, j)* is either in cache or moved to main memory
- At most 2**M* live *c(i, j)* in phase *p*
- At most 2**M* distinct elements of *A* in cache during phase *p*; same for B
  - Either in cache at beginning or moved to cache during phase
  - Let A_p be set of elements in cache during phase *p*; same for B_p

How many multiplies in phase p?

- Let $S_{p,+}$ = set of rows of A with $M^{1/2}$ or more elements in A_p
- Let $S_{p,-}$ = set of rows of A with fewer
- $|S_{p,+}| \leq 2{*}M^{1/2}$
- Consider rows in $S_{p,+}$:
  - Operation "$a(i, :) \times B$" touches each element of B only once
  - So, no. of scalar multiplies $\leq |S_{p,+}| * (2{*}M) = 4{*}M^{3/2}$
- For rows in $S_{p,-}$, consider that "$c(i,j)$ = row x col"
  - Thus, (# multiplies) ≤ (no. live) x (max row len) ≤ $2{*}M^{3/2}$

Final bound on multiplies

$$\begin{aligned}
\text{Total no. of multiplies} &= n^3 \\
\text{No. of multiplies per phase} &\leq 6M^{\frac{3}{2}} \\
\text{No. of phases} &\geq \left\lceil \frac{n^3}{6M^{\frac{3}{2}}} \right\rceil \\
\text{Total no. of words transferred} &\geq M \cdot \left(\frac{n^3}{6M^{\frac{3}{2}}} - 1 \right) \\
&= \frac{n^3}{6\sqrt{M}} - M
\end{aligned}$$

Can we do better? Nope.

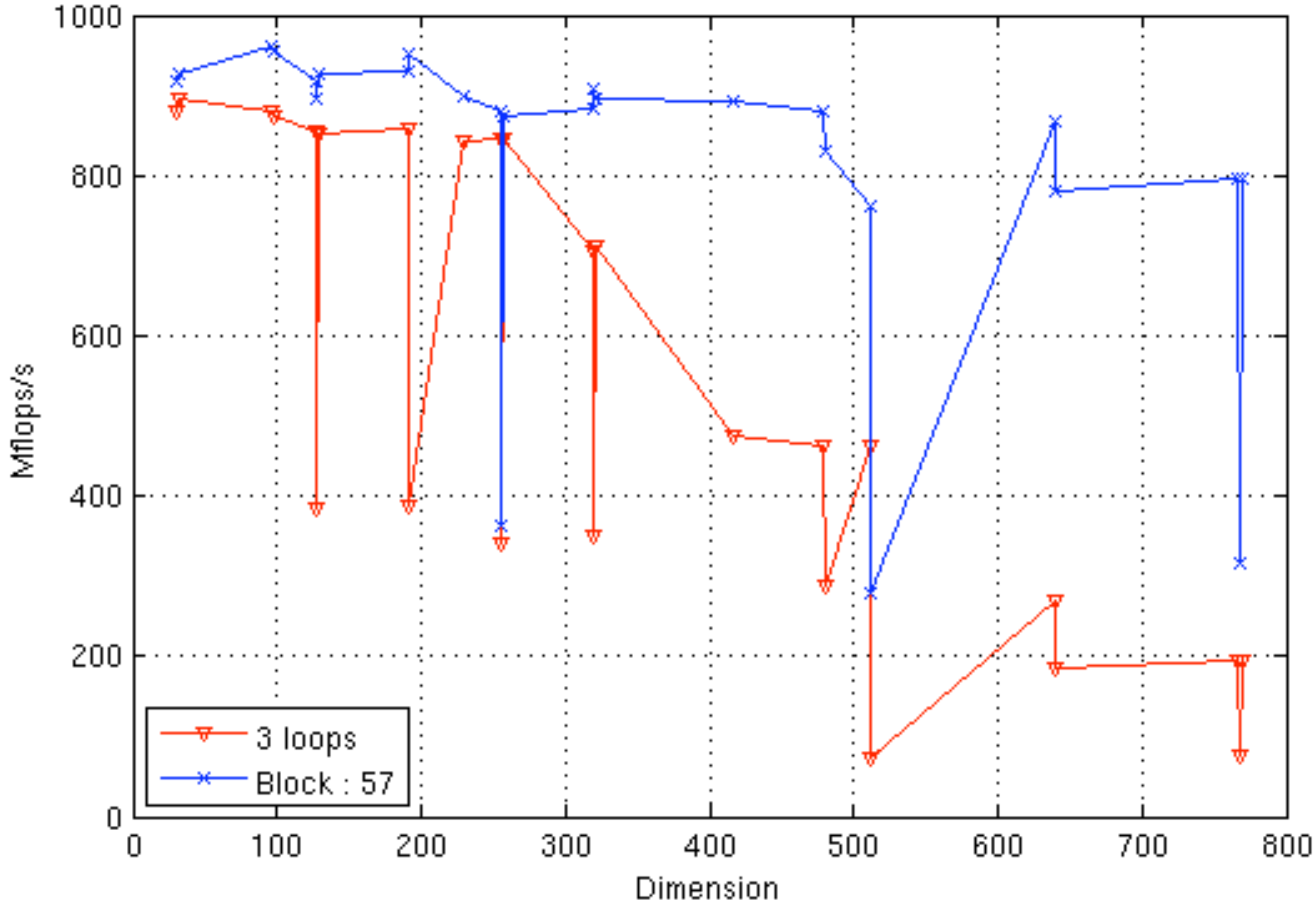
- **Theorem** [Hong and Kung (1981)]: Any schedule of conventional matrix multiply must transfer $\Omega(n^3 / \sqrt{M})$ words between slow and fast memory, where $M < n^2 / 6$.
- We did intuitive proof by Toledo (1999)
- Historical note: Rutledge & Rubinstein (1951—52)
- So cached block matrix multiply is **asymptotically optimal**.

$$b = O\left(\sqrt{M}\right) \implies m = O\left(\frac{n^3}{b}\right) = O\left(\frac{n^3}{\sqrt{M}}\right)$$

What happens in practice?

- Experiment: One-level cache-blocked matrix multiply
- Block size chosen as square, by exhaustive search over sizes up to 64

Tiled MM on AMD Opteron 2.2 GHz (4.4 Gflop/s peak), 1 MB L2 cache



We evidently still have a lot of work to do...

Administrivia

Two joint classes with CS 8803 SC

- **Tues 2/19**: Floating-point issues in parallel computing by me
- **Tues 2/26**: GPGPUs by Prof. Hyesoon Kim
  - **Scribe?**
- **Both classes meet in Klaus 1116E**

Homework 1:
Parallel conjugate gradients

- **Extension**: Due Wednesday 2/27 @ 8:30 am
- Implement a parallel solver for Ax = b (serial C version provided)
  - Evaluate on three matrices: 27-pt stencil, and two application matrices
  - “Simplified:” No preconditioning
- **Performance models to understand scalability of your implementation**
  - Make measurements
  - Build predictive models
- Collaboration encouraged: Compare programming models or platforms

Administrative stuff

- **New room** (dumpier, but cozier?): College of Computing Building **(CCB) 101**
- **Accounts**: Apparently, you already have them
- Front-end login node: **ccil.cc.gatech.edu** (CoC Unix account)
  - We "own" **warp43—warp56**
  - Some docs (**MPI**): http://www-static.cc.gatech.edu/projects/ihpcl/mpi.html
  - **Sign-up** for mailing list: https://mailman.cc.gatech.edu/mailman/listinfo/ihpc-lab

Projects

- Your goal should be to do something useful, interesting, and/or publishable!
  - Something you're already working on, suitably adapted for this course
  - Faculty-sponsored/mentored
  - Collaborations encouraged

My criteria for "approving" your project

- "Relevant to this course:" Many themes, so think (and "do") broadly
  - Parallelism and architectures
  - Numerical algorithms
  - Programming models
  - Performance modeling/analysis

General styles of projects

- Theoretical: Prove something hard (high risk)
- Experimental:
  - Parallelize something
  - Take existing parallel program, and improve it using models & experiments
  - Evaluate algorithm, architecture, or programming model

Examples

- *Anything of interest to a faculty member/project outside CoC*
- Parallel sparse triple product ($R*A*R^T$, used in multigrid)
- Future FFT
- Out-of-core or I/O-intensive data analysis and algorithms
- Block iterative solvers (convergence & performance trade-offs)
- Sparse LU
- Data structures and algorithms (trees, graphs)
- Discrete-event approaches to continuous systems simulation
- Automated performance analysis and modeling, tuning
- “Unconventional,” but related
  - Distributed deadlock detection for MPI
  - UPC language extensions (dynamic block sizes)
  - Exact linear algebra

“In conclusion…”

Backup slides