

Lecture 3

Vectorization

Memory system optimizations

Performance Characterization

Announcements

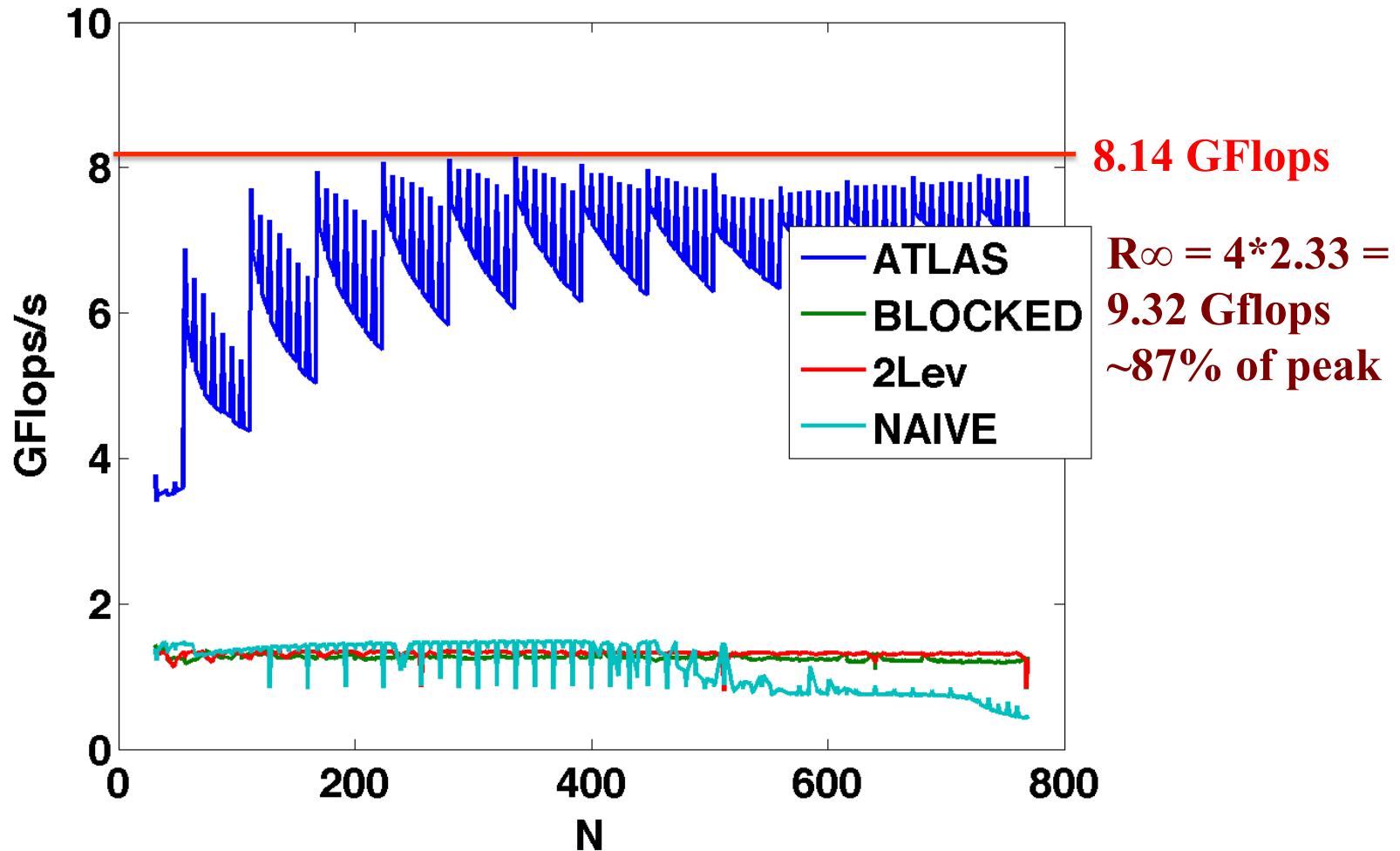
- Submit NERSC form
- Login to lilliput.ucsd.edu using your AD credentials?

Today's Lecture

- Vectorization and SSE
- Other optimizations
- Performance
 - ◆ Measurement
 - ◆ Metrics

Assignment #1

- Blocking for cache will boost performance but a lot more is needed to approach ATLAS' performance



Required performance programming

- Hierarchical blocking
 - ◆ Multiple levels of cache and/or TLB
 - ◆ Cache friendly layouts
 - ◆ Register blocking (with unrolling)
- SSE intrinsics
- Autotuning
 - ◆ Computer generated variants & blocking factors
 - ◆ PHiPAC → ATLAS, in Matlab
 - ◆ Performance models not sufficiently accurate
 - ◆ Need to tune to matrix size
- See Jim Demmel's lecture
www.cs.berkeley.edu/~demmel/cs267_Spr12/Lectures/lecture02_memhier_jwd12.ppt

Blocking for cache: motivation

- Assume a 2 level memory hierarchy, fast and slow
- Minimum running time = $f \times \gamma$ when all data is in fastest memory, f =# arithmetic ops, γ = time / flop

- Actual time

$$f \times \gamma + m \times t_m = f \times \gamma \times (1 + t_m/\gamma \times 1/q)$$

m = # words transferred

t_m = slow memory access time

$q = f/m$ = computational intensity

t_m/γ Establishes machine balance

- What is the computational intensity for matrix multiply?

Blocking for cache: operational details

- For matrix multiply, all 3 blocks must fit into cache
- If M_{fast} = size of fast memory

$$3b^2 \leq M_{\text{fast}} \Rightarrow q \approx b \leq (M_{\text{fast}}/3)^{1/2}$$

- To run at half peak speed

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

- We change the order of arithmetic, so slightly different answers due to roundoff

- **Lower bound Theorem (Hong & Kung, 1981):**

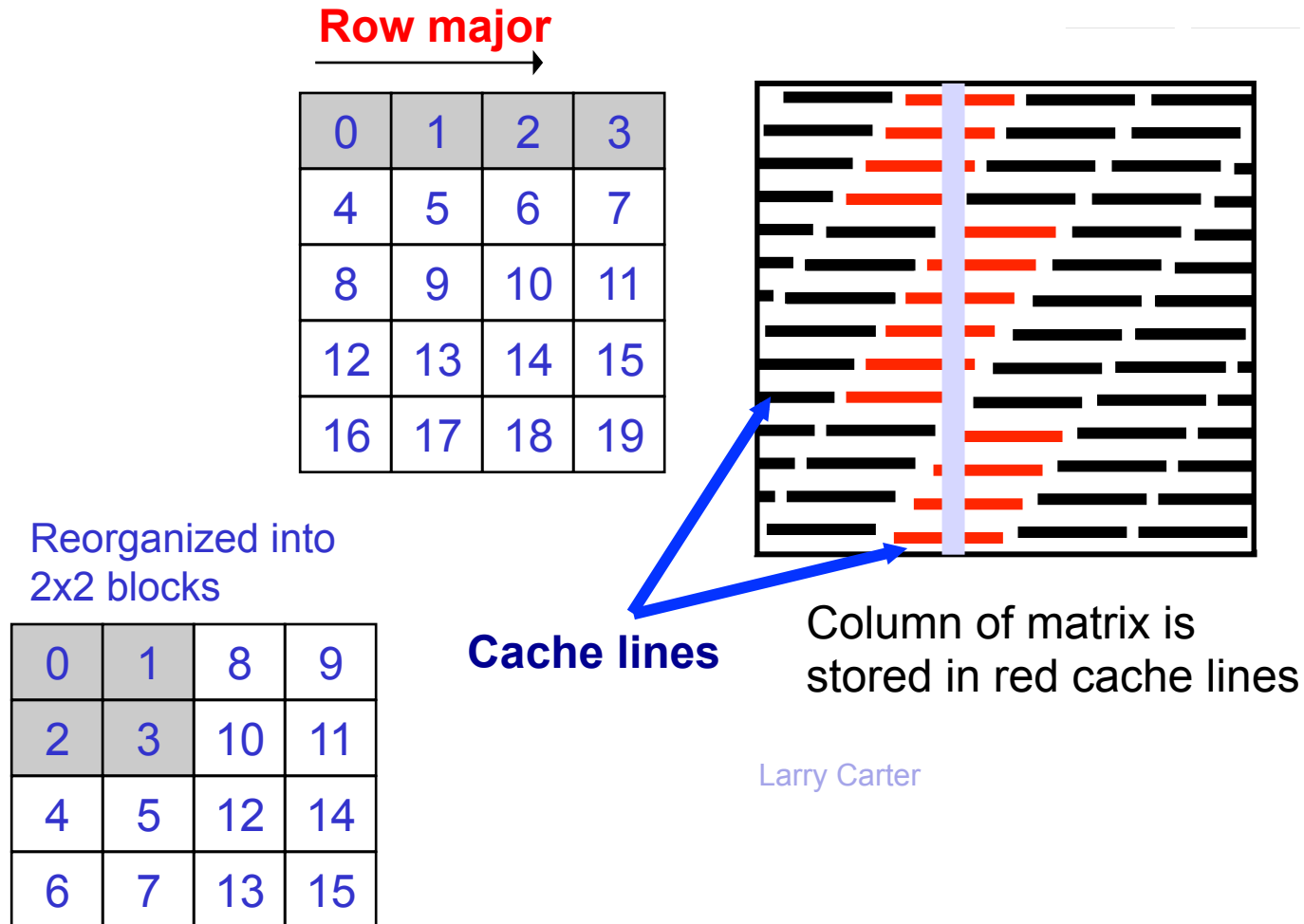
Any reorganization of this algorithm (that uses only associativity) is limited to

$$q = O((M_{\text{fast}})^{1/2})$$

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

Cache interference

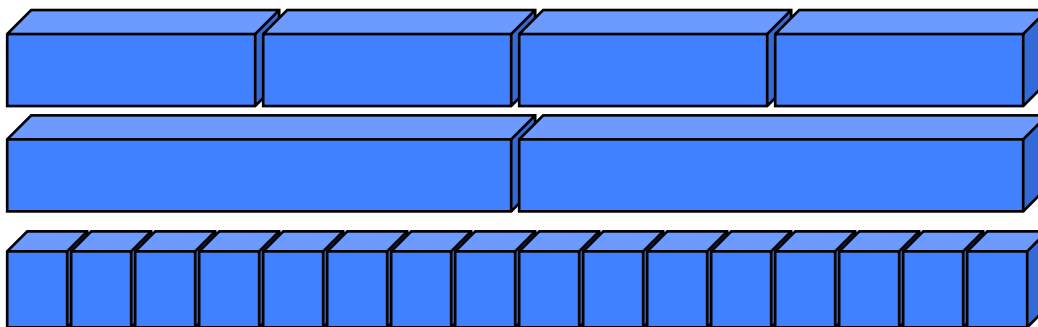
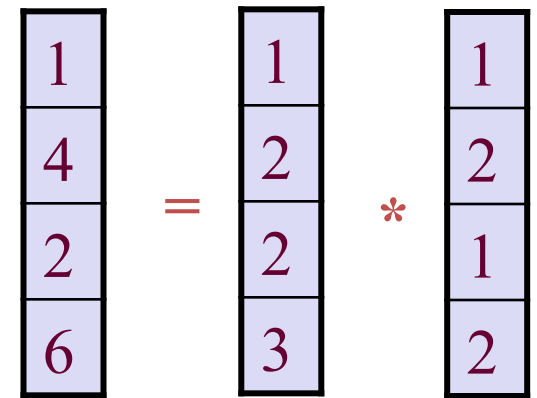
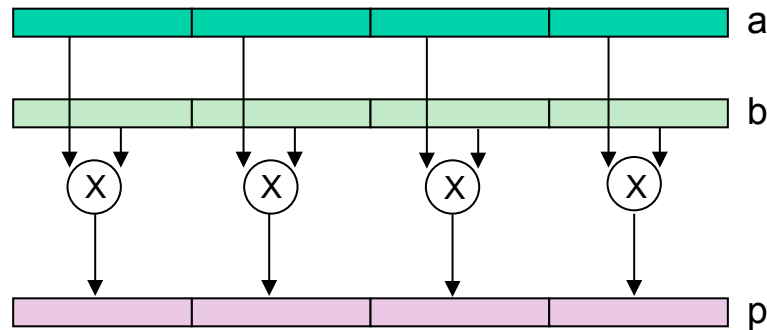
- Copying and cache friendly layouts



Streaming SIMD Extensions

- SIMD instruction set on short vectors
- SSE (SSE3 on Bang, SSE4 on Intel Nehalem)
- On Bang: 16x128 bit vector registers

for $i = 0:N-1$ { $p[i] = a[i] * b[i];$ }

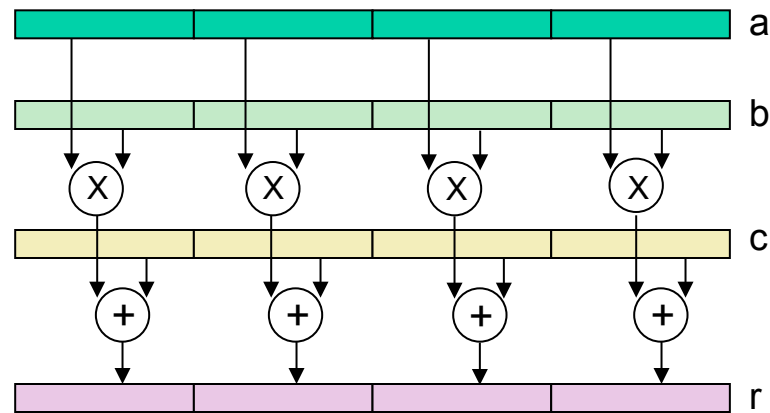


4 floats/ints

2 doubles

16 bytes

Fused Multiply/Add



$$r[0:3] = c[0:3] + a[0:3]*b[0:3]$$

Courtesy of Mercury Computer Systems, Inc.

How do we use the SSE instructions?

- Low level: assembly language or libraries
- Higher level: a vectorizing compiler

`gcc -O3 -ftree-vectorizer-verbose=2 s35.c`

```
int N;  
float a[N], b[N], c[N];  
    for (int i=0; i<N; i++)  
        a[i] = b[i] + c[i];
```

`s35.c:8: note: LOOP VECTORIZED.`

`s35.c:4: note: vectorized 1 loops in function`

How does non-vectorized code compare?

- Low level: assembly language or libraries
- Higher level: a vectorizing compiler

```
#pragma novector
```

```
for (int i=0; i<N; i++) // N = 2048 * 1024
```

```
    a[i] = b[i] + c[i];
```

Single precision, running on a Nehalem processor using pgcc

With vectorization : 7.693 sec.

Without vectorization : 10.17 sec.

- Double precision

With vectorization : 11.88 sec.

Without vectorization : 11.85 sec.

How does the vectorizer work?

- Transformed code

```
for (i = 0; i < 1024; i+=4)
    a[i:i+3] = b[i:i+3] + c[i:i+3];
```

- Vector instructions

```
for (i = 0; i < 1024; i+=4){
    vB = vec_ld( &b[i] );
    vC = vec_ld( &c[i] );
    vA = vec_add( vB, vC );
    vec_st( vA, &a[i] );
}
```

What prevents vectorization

- Data dependencies

```
for (int i = 1; i < N; i++)
```

```
    b[i] = b[i-1] + 2;
```

```
b[1] = b[0] + 2;
```

```
b[2] = b[1] + 2;
```

```
b[3] = b[2] + 2;
```

Loop not vectorized: data dependency

- Inner loops only

```
for(int j=0; j< reps; j++)
```

```
    for (int i=0; i<N; i++)
```

```
        a[i] = b[i] + c[i];
```

What prevents vectorization

- Interrupted flow out of the loop

```
for (i=0; i<n; i++) {  
    a[i] = b[i] + c[i];  
    maxval = (a[i] > maxval ? a[i] : maxval);  
    if (maxval > 1000.0) break;  
}
```

Loop not vectorized/parallelized: multiple exits

- This loop will vectorize

```
for (i=0; i<n; i++) {  
    a[i] = b[i] + c[i];  
    maxval = (a[i] > maxval ? a[i] : maxval);  
}
```

Run-time data dependence testing

- Restrict keyword needed to ensure correct semantics
<http://www.devx.com/tips/Tip/13825>
“During the scope of the pointer declaration, all data accessed through it will be accessed .. through any other pointer... [thus] a given object cannot be changed through another pointer.”

`icpc -O2 -vec-report3 -restrict t2a.c`

```
void copy_conserve(char *restrict p, char *restrict q, int n) {  
    int i;  
    if (p+n < q || q+n < p)  
        #pragma ivdep  
        for (i = 0; i < n; i++) p[i] = q[i];      /* vector loop */  
    else  
        for (i = 0; i < n; i++) p[i] = q[i];      /* serial loop */  
}
```

`copy.c(11): (col. 3) remark: LOOP WAS VECTORIZED.`

C++ intrinsics

- The compiler may not be able to handle all situations, such as conditionals
- We'll need to use library intrinsics, that map directly onto machine instructions (one or more)
- These are supported by gcc and other compilers
- The interface provides 128 bit data types and operations on those datatypes
- Data may need to be aligned

SSE Pragmatics

- 16 XMM data registers (128 bit)
(Don't use the MMX 64 bit registers)
- Vector operations (add, subtract, etc)
- Data transfer (load/store)
- Shuffling (handles conditionals)
- May need to invoke compiler options depending on level of optimization

Blocking for registers in matrix multiply

- We can apply blocking to the registers, too
- Registers are a scarce resource
- 2x2 matrix multiply
- Store array values on the stack

$$\begin{array}{l} C_{00} += A_{00}B_{00} + A_{01}B_{10} \\ C_{10} += A_{10}B_{00} + A_{11}B_{10} \\ C_{01} += A_{00}B_{01} + A_{01}B_{11} \\ C_{11} += A_{10}B_{01} + A_{11}B_{11} \end{array} \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

Rewrite as SIMD algebra

```
C00_C01 += A00_A00 * B00_B01
C10_C11 += A10_A10 * B00_B01
C00_C01 += A01_A01 * B10_B11
C10_C11 += A11_A11 * B10_B11
```

2x2 Matmul with SSE intrinsics

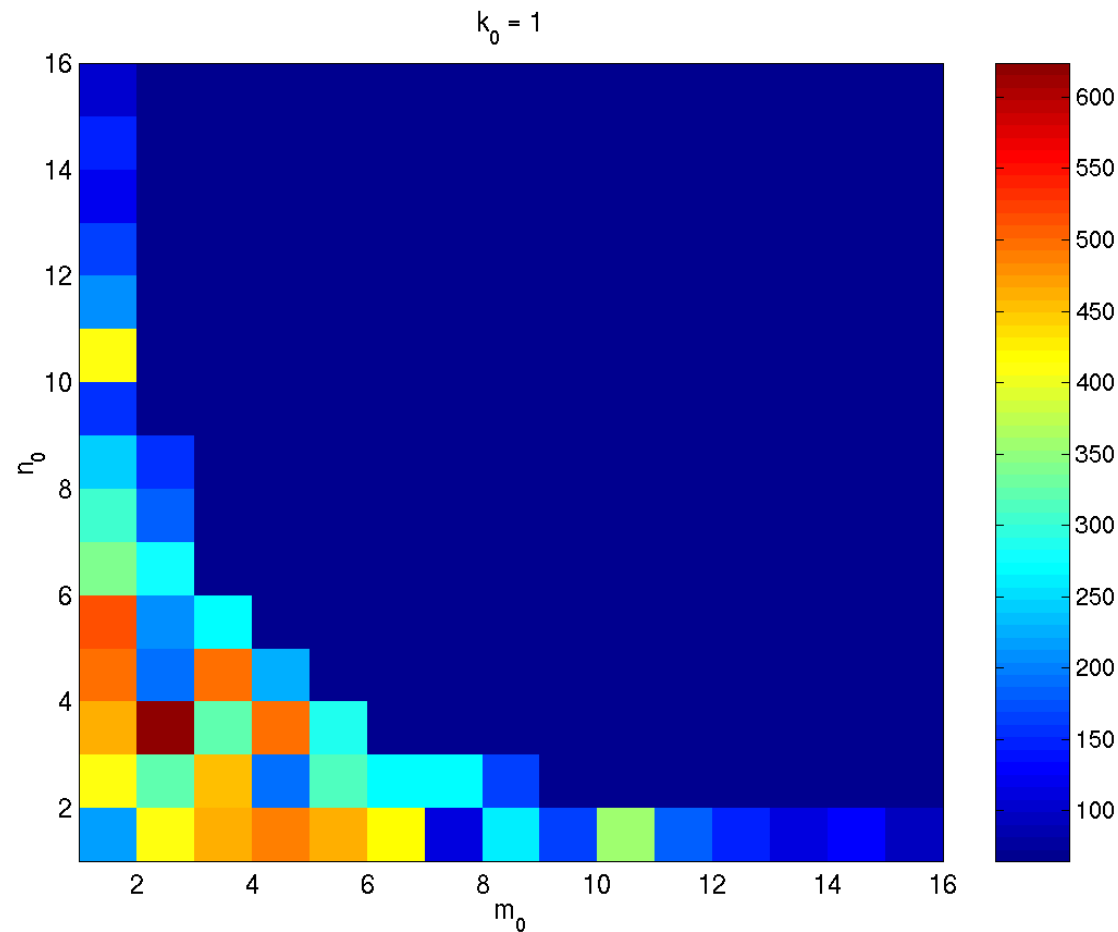
```
#include <emmintrin.h>

void square_dgemm (int N, double* A, double* B, double* C){
  __m128d c1 = _mm_loadu_pd( C+0*N);    //load unaligned block in C
  __m128d c2 = _mm_loadu_pd( C+1*N);
  for( int i = 0; i < 2; i++ ){
    __m128d a1 = _mm_load1_pd( A+i*0*N); //load i-th column of A (A0x,A0x)
    __m128d a2 = _mm_load1_pd( A+i*1*N); //load i-th column of A (A1x,A1x)
    __m128d b = _mm_load_pd( B+i*N);    //load aligned i-th row of B
    c1 = _mm_add_pd( c1, _mm_mul_pd( a1, b ) ); //rank-1 update
    c2 = _mm_add_pd( c2, _mm_mul_pd( a2, b ) );
  }
  _mm_storeu_pd( C+0*N, c1 );           //store unaligned block in C
  _mm_storeu_pd( C+1*N, c2 );
}
```

```
C00_C01 += A00_A00 * B00_B01
C10_C11 += A10_A10 * B00_B01
C00_C01 += A01_A01 * B10_B11
C10_C11 += A11_A11 * B10_B11
```

$$\begin{pmatrix} A00 & A01 \\ A10 & A11 \end{pmatrix} \begin{pmatrix} B00 & B01 \\ B10 & B11 \end{pmatrix}$$

A search space



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

Jim Demmel

Loop Unrolling

- Common loop optimization strategy
- Duplicate the body of the loop

```
for (int i=0; i < n ; i++)  
    z[i] = x[i] + y[i];
```

```
for (int i=0; i < n ; i++4){  
    z[i+0] = x[i+0] + y[i+0];  
    z[i+1] = x[i+1] + y[i+1];  
    z[i+2] = x[i+2] + y[i+2];  
    z[i+3] = x[i+3] + y[i+3];  
}
```

- Register utilization, instruction scheduling
- May be combined with “jamming:”
unroll and jam
- Not always advantageous

Today's Lecture

- Vectorization and SSE
- Performance
 - ◆ Measurement
 - ◆ Metrics

Challenges to measuring performance

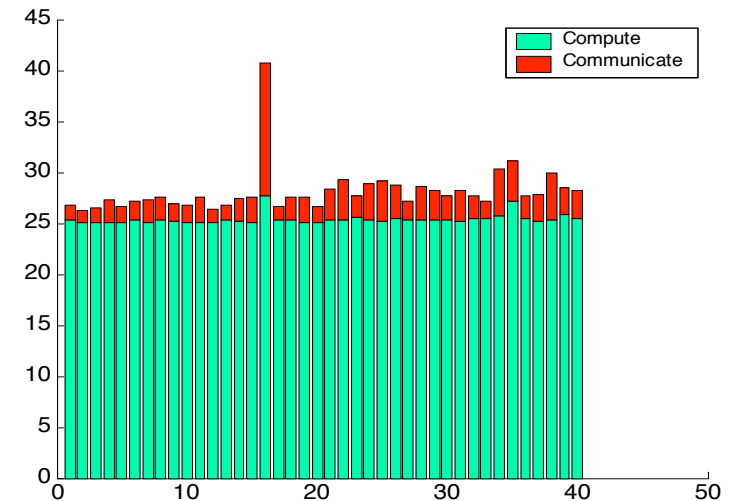
- Reproducibility
 - ◆ Transient system operating conditions
 - ◆ Differing systems or program configuration
- Measurements are imprecise
 - ◆ “Heisenberg uncertainty principle:”
measurement technique may affect performance
 - ◆ Overheads and inaccuracy
- Explain anomalous behavior, but ignore anomalies that are not significant

Complications

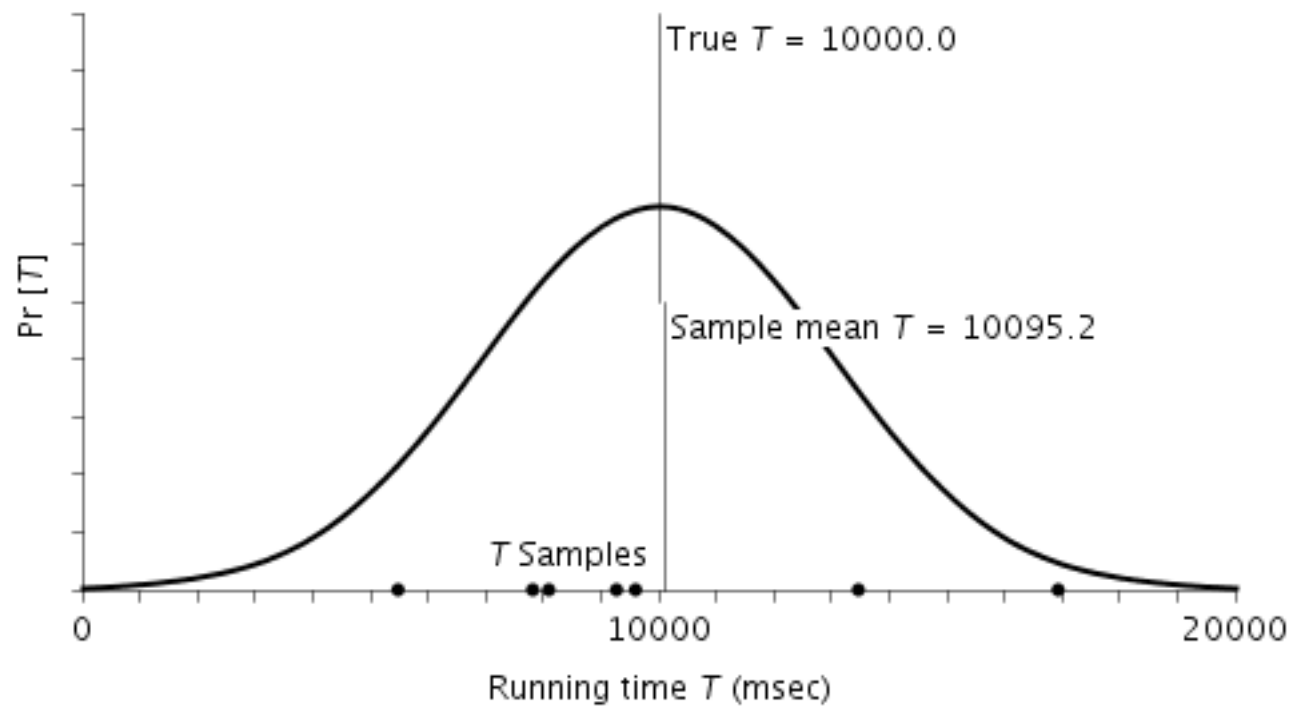
- Cost of measuring a full run is prohibitive
 - ♦ Ignore startup code if you plan to run for a much longer time in production
- Transient behavior
 - ♦ Repeat your measurements
 - ♦ “Warm up” the code before collecting measurements
 - ♦ Ignore outliers unless their behavior is important to you
 - ♦ Average time, maximum time, minimum time?

Measurement collection

- Report the *best* timings
 - ▶ Repeat results $\times 3$ to 5 until at least 2 measures agree to within... 5%, 10%
 - ▶ Report the minimum time
- Also report outliers
- A scatter plot or error bar can be useful

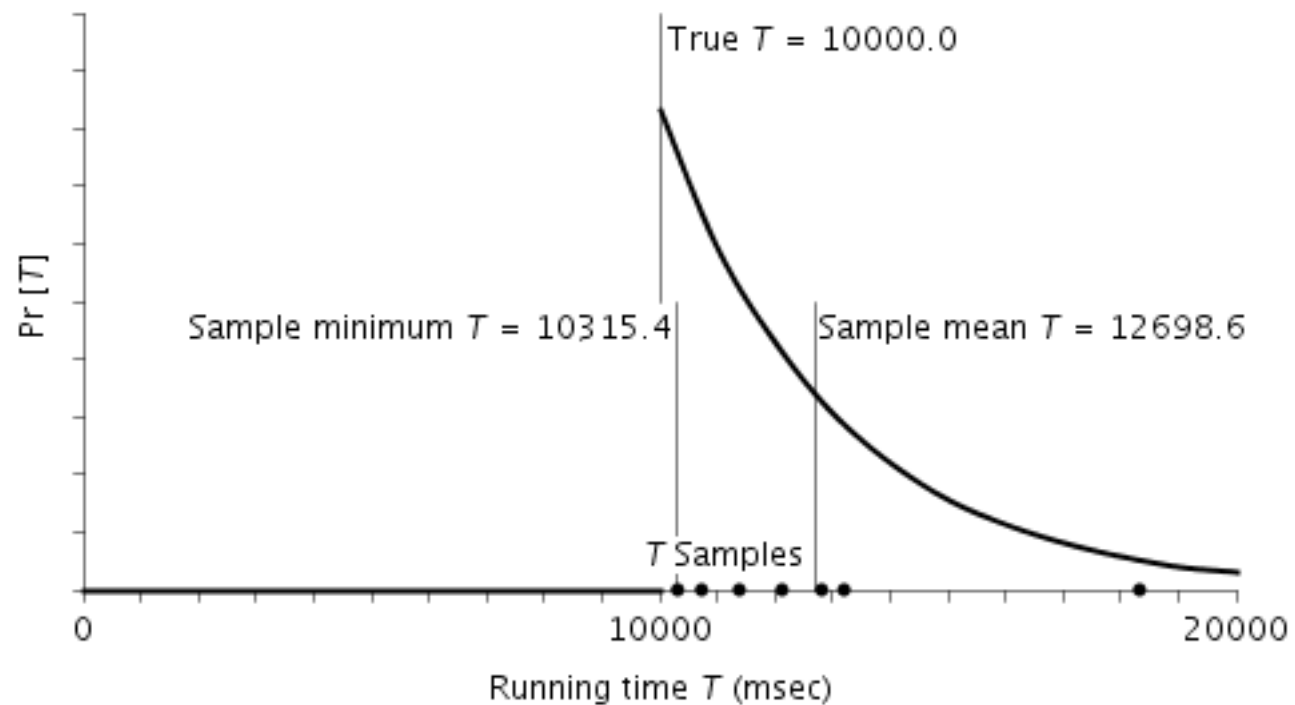


Why do we take the minimum time?



Alan Kaminsky. *Building Parallel Programs: SMPs, Clusters, and Java*. Copyright © 2010 Course Technology

Measurement errors are not distributed symmetrically



Timing collection

- Measures of time
 - ▶ Elapsed, or “wall clock” time
 - ▶ CPU time = system + user time
 - ▶ Overhead, resolution, and quantization effects
- Measurement tools
 - ▶ Can be platform dependent, especially library routines
 - ▶ Unix `time` command does a reasonable job for long-running programs
 - ▶ `gettimeofday()`

Enable others to reproduce your results

- Builds confidence within a community
- Report where you ran, software versions, processor, etc.
 - ▶ `uname -a`
 - ▶ Linux lilliput 2.6.35-30-server #61-Ubuntu SMP Tue Oct 11 18:09:44 UTC 2011 x86_64 GNU/Linux
 - ▶ `gcc --version`
 - ▶ gcc version 4.4.5 (Ubuntu/Linaro 4.4.4-14ubuntu5)
 - ▶ `icpc --version`
 - ▶ icpc (ICC) 12.0.2 20110112
 - ▶ `nvcc --version`
 - ▶ Cuda compilation tools, release 4.0, V0.2.1221
 - ▶ Access processor configuration information
 - ▶ Device # 0 has 30 cores
 - ▶ Device # 1 has 4 cores
 - ▶ Choosing device 0
 - ▶ Device is a GeForce GTX 285, capability: 1.3
 - ▶ CUDA Driver version: 2030, runtime version: 2030

Fin