Logistics

- Lecture notes and slides for first two lectures are up: http://www.cs.cornell.edu/~bindel/class/cs5220-s10.
- You should receive cluster information soon for crocus.csuglab.cornell.edu. When you do, make sure you can log in!
- We will be setting up a wiki for the class — among other things, this will be a way to form groups.
- Hope to have the first assignment ready by Wednesday.
Consider naive square matrix multiplication:

```c
#define A(i,j) AA[j*n+i]
#define B(i,j) BB[j*n+i]
#define C(i,j) CC[j*n+i]

for (i = 0; i < n; ++i) {
    for (j = 0; j < n; ++j) {
        C(i,j) = 0;
        for (k = 0; k < n; ++k)
            C(i,j) += A(i,k)*B(k,j);
    }
}
```

How fast can this run?
Why matrix multiply?

- Key building block for dense linear algebra
- Same pattern as other algorithms (e.g. transitive closure via Floyd-Warshall)
- Good model problem (well studied, illustrates ideas)
- Easy to find good libraries that are hard to beat!
1000-by-1000 matrix multiply on my laptop

- Theoretical peak: 10 Gflop/s using both cores
- Naive code: 330 MFlops (3.3% peak)
- Vendor library: 7 Gflop/s (70% peak)

Tuned code is $20 \times$ faster than naive!
Simple model

Consider two types of memory (fast and slow) over which we have complete control.

- $m = \text{words read from slow memory}$
- $t_m = \text{slow memory op time}$
- $f = \text{number of flops}$
- $t_f = \text{time per flop}$
- $q = f/m = \text{average flops / slow memory access}$

Time:

$$ft_f + mt_m = ft_f \left(1 + \frac{t_m}{t_f} \frac{1}{q}\right)$$

Two important ratios:

- $t_m/t_f = \text{machine balance (smaller is better)}$
- $q = \text{computational intensity (larger is better)}$
How big can $q$ be?

1. Dot product: $n$ data, $2n$ flops
2. Matrix-vector multiply: $n^2$ data, $2n^2$ flops
3. Matrix-matrix multiply: $2n^2$ data, $2n^2$ flops

These are examples of level 1, 2, and 3 routines in *Basic Linear Algebra Subroutines* (BLAS). We like building things on level 3 BLAS routines.
$q$ for naive matrix multiply

$q \approx 2$ (on board)
Better locality through blocking

Basic idea: rearrange for smaller working set.

```c
for (I = 0; I < n; I += bs) {
    for (J = 0; J < n; J += bs) {
        block_clear(&(C(I,J)), bs, n);
        for (K = 0; K < n; K += bs)
            block_mul(&(C(I,J)), &(A(I,K)), &(B(K,J)),
                       bs, n);
    }
}
```

Q: What do we do with “fringe” blocks?
$q$ for naive matrix multiply

$q \approx b$ (on board). If $M_f$ words of fast memory, $b \approx \sqrt{M_f/3}$.

Th: (Hong/Kung 1984, Irony/Tishkin/Toledo 2004): Any reorganization of this algorithm that uses only associativity and commutativity of addition is limited to $q = O(\sqrt{M_f})$

Note: Strassen uses distributivity...
Better locality through blocking

Timing for matrix multiply

- Naive
- Blocked
- DSB
Truth in advertising
Recursive blocking

- Can use blocking idea recursively (for L2, L1, registers)
- Best blocking is *not* obvious!
- Need to tune bottom level carefully...
Idea: Cache-oblivious algorithms

Index via Z-Morton ordering ("space-filling curve")

- Pro: Works well for any cache size
- Con: Expensive index calculations

Good idea for ordering meshes?
Copy optimization

Copy blocks into contiguous memory

- Get alignment for SSE instructions (if applicable)
- Unit stride even across bottom
- Avoid *conflict* cache misses
Auto-tuning

Several different parameters:

▶ Loop orders
▶ Block sizes (across multiple levels)
▶ Compiler flags?

Use automated search!

Idea behind ATLAS (and earlier efforts like PhiPAC).
My last matrix multiply

- Good compiler (Intel C compiler) with hints involving aliasing, loop unrolling, and target architecture. Compiler does auto-vectorization.
- L1 cache blocking
- Copy optimization to aligned memory
- Small \((8 \times 8 \times 8)\) matrix-matrix multiply kernel found by automated search. Looped over various size parameters.

On that machine, I got 82% peak. Here... less than 50% so far.
“We should forget bout small efficiencies, say about 97% of the time: premature optimization is the root of all evil.”
– C.A.R. Hoare (quoted by Donald Knuth)

- Best case: good algorithm, efficient design, obvious code
- Tradeoff: speed vs readability, debuggability, maintainability...
- Only optimize when needful
- Go for low-hanging fruit first: data layouts, libraries, compiler flags
- Concentrate on the bottleneck
- Concentrate on inner loops
- Get correctness (and a test framework) first
Tuning tip 0: use good tools

- We have gcc. The Intel compilers are better.
- Fortran compilers often do better than C compilers (less aliasing)
- Intel VTune, cachegrind, and Shark can provide useful profiling information (including information about cache misses)
Tuning tip 1: use libraries!

- Tuning is painful! You will see...
- Best to build on someone else’s efforts when possible
Tuning tip 2: compiler flags

- `-O3`: Aggressive optimization
- `-march=core2`: Tune for specific architecture
- `-ftree-vectorize`: Automatic use of SSE (supposedly)
- `-funroll-loops`: Loop unrolling
- `-ffast-math`: Unsafe floating point optimizations

Sometimes *profiler-directed* optimization helps. Look at the gcc man page for more.
Tuning tip 3: Attend to memory layout

- Arrange data for unit stride access
- Arrange algorithm for unit stride access!
- Tile for multiple levels of cache
- Tile for registers (loop unrolling + “register” variables)
Tuning tip 4: Use small data structures

- Smaller data types are faster
  - Bit arrays vs int arrays for flags?
  - Minimize indexing data — store data in blocks
  - Some advantages to *mixed precision* calculation (*float* for large data structure, *double* for local calculation) — more later in the semester!

- Sometimes recomputing is faster than saving!
Tuning tip 5: Inline judiciously

- Function call overhead *often* minor...
- ... but structure matters to optimizer!
- C++ has `inline` keyword to indicate inlined functions
Tuning tip 6: Avoid false dependencies

Arrays in C can be aliased:

\[
\begin{align*}
a[i] &= b[i] + c; \\
a[i+1] &= b[i+1] \times d;
\end{align*}
\]

Can't reorder – what if \(a[i+1]\) refers to \(b[i]\)? But:

\[
\begin{align*}
float b1 &= b[i]; \\
float b2 &= b[i+1]; \\
a[i] &= b1 + c; \\
a[i+1] &= b2 \times d;
\end{align*}
\]

Declare no aliasing via restrict pointers, compiler flags, pragmas...
Tuning tip 7: Beware inner loop branches!

- Branches slow down code if hard to predict
- May confuse optimizer that only deals with basic blocks
Tuning tip 8: Preload into local variables

```c
while (...) {
    *res++ = filter[0]*signal[0] +
             filter[1]*signal[1] +
             filter[2]*signal[2];
    signal++;
}
```
Tuning tip 8: Preload into local variables

... becomes

```c
float f0 = filter[0];
float f1 = filter[1];
float f2 = filter[2];
while (...) {
    *res++ = f0*signal[0] +
            f1*signal[1] +
            f2*signal[2];
    signal++;
}
```
Tuning tip 9: Loop unrolling plus software pipelining

```c
float s0 = signal[0], s1 = signal[1],
    s2 = signal[2];
*res++ = f0*s0 + f1*s1 + f2*s2;
while (...) {
    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;
}

Note: more than just removing index overhead!
Remember: -funroll-loops!
```
Tuning tip 10: Expose independent operations

- Use local variables to expose independent computations
- Balance instruction mix for different functional units

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

What to use for high performance?

- Function calculation or table of precomputed values?
- Several (independent) passes over a data structure or one combined pass?
- Parallel arrays vs array of records?
- Dense matrix vs sparse matrix (only nonzeros indexed)?
- MATLAB vs C for dense linear algebra codes?
Your assignment (out Weds)

- Learn to log into cluster.
- Find someone to work with (wiki should help? assigned?)
- Optimize square matrix-matrix multiply.

Details and pointers to resources in next couple days.