CS 294-73
Software Engineering for Scientific Computing

Lecture 15: Development for Performance

Performance

- How fast does your code run ?
- How fast can your code run ?
- How fast can your algorithm run ?
- How do you make your code run as fast as possible ?
  - What is making it run more slowly than the algorithm permits ?

Performance Loop

- Programming to a cartoon ("model") for how you're your machine behaves.
- Measuring the behavior of your code.
- Modifying your code to improve performance.
- When do you stop ?

Naïve vs. Vendor DGEMM Bounds Expectations

```
>./naive.exe
```
Two flops / 2+ doubles read/written

```
n 31, MFlop/sec = 2018.29
n 32, MFlop/sec = 1754.92
n 96, MFlop/sec = 1746.74
n 97, MFlop/sec = 1906.88
n 127, MFlop/sec = 1871.38
n 128, MFlop/sec = 1674.05
n 129, MFlop/sec = 1951.06
n 191, MFlop/sec = 1673.44
n 192, MFlop/sec = 1514.24
n 229, MFlop/sec = 1915.5
n 255, MFlop/sec = 1692.96
n 256, MFlop/sec = 827.36
n 257, MFlop/sec = 1751.56
n 319, MFlop/sec = 1762.5
n 320, MFlop/sec = 1431.29
n 321, MFlop/sec = 1714.46
n 479, MFlop/sec = 1569.42
n 480, MFlop/sec = 1325.46
n 511, MFlop/sec = 1242.37
n 512, MFlop/sec = 645.815
n 639, MFlop/sec = 247.698
n 640, MFlop/sec = 231.998
n 767, MFlop/sec = 211.702
n 768, MFlop/sec = 221.34
n 769, MFlop/sec = 204.241
```

```
>./blas.exe
```
M flops / double (“speed of light”)

```
n 31, MFlop/sec = 8828.4
n 32, MFlop/sec = 11479.1
n 96, MFlop/sec = 17448.5
n 97, MFlop/sec = 14472.2
n 127, MFlop/sec = 15743.9
n 128, MFlop/sec = 16956.6
n 129, MFlop/sec = 19335.8
n 191, MFlop/sec = 25332.7
n 192, MFlop/sec = 26786
n 229, MFlop/sec = 27853.2
n 255, MFlop/sec = 28101
n 256, MFlop/sec = 30022.1
n 257, MFlop/sec = 28344.9
n 319, MFlop/sec = 28477
n 320, MFlop/sec = 28783.5
n 321, MFlop/sec = 28163.6
n 479, MFlop/sec = 29673.5
n 480, MFlop/sec = 30142.8
n 511, MFlop/sec = 29283.7
n 512, MFlop/sec = 30681.8
n 639, MFlop/sec = 28603.6
n 640, MFlop/sec = 31517.6
n 767, MFlop/sec = 29292.7
n 768, MFlop/sec = 31737.5
n 769, MFlop/sec = 29681.4
```

Naïve falls off a cliff for large matrices.

$$C_{i,j} = \sum_{k=1}^{N} A_{i,k} B_{k,j}$$

- Column-wise storage -> Access to B is stride N. As N gets large, you increasingly frequent L2 cache misses.

Premature optimization

- Otherwise known as the root of all evil
- Your first priority with a scientific computing code is correctness.
  - A buggy word-processor might be acceptable if it is still responsive.
  - A buggy computer model is not an acceptable scientific tool
- Highly optimized code can be difficult to debug.
  - If you optimize code, keep the unoptimized code available as an option.

...but you can't completely ignore performance

- Changing your data structures late in the development process can be very troublesome
  - Unless you have isolated that design choice with good modular design
- Changing your algorithm choice after the fact pretty much puts you back to the beginning.
- So, the initial phase of development is:
  - make your best guess at the right algorithm
  - make your best guess at the right data structures
    - What is the construction pattern ?
    - What is the access pattern ?
    - How often are you doing either one ?
  - Insulate yourself from the effects of code changes with encapsulation and interfaces.

Key step in optimization: Measurement

- It is amazing the number of people that start altering their code for performance based on their own certainty of what is running slowly.
  - Mostly they remember when they wrote some particularly inelegant routine that has haunted their subconscious.
- The process of measuring code run time performance is called *profiling*.   Tools to do this are called *profilers*.
- It is important to measure the right thing
  - Does your input parameters reflect the case you would like to run fast?
  - Don’t measure code compiled with the debug flag “-g”
    - You use the optimization flags “-O2” or “-O3”
    - For that last 5% performance improvement from the compiler you have a few dozen more flags you can experiment with
    - You do need to verify that your “-g” code and your “-O3” code get the same answer.
      - some optimizations alter the strict floating-point rules

Profilers

- *Sampling profilers* are programs that run while your program is running and *sample the call stack*
  - sampling the call stack is like using the 'where' or 'backtrace' command in gdb.
  - This sampling is done at some pre-defined regular interval
    - perhaps every millisecond
  - Advantages
    - The profiling does little to disturb the thing it is measuring.
      - The caveat there is not sampling too often
    - Detailed information about the state of the processor at that moment can be gathered
  - Disadvantages
    - No reporting of call counts
      - is this one function that runs slowly, or a fast function that is called a lot of times ? what course of action is appropriate ?
    - oversampling will skew measurement

Some examples of Sampling Profilers

- Apple
  - Shark (older Xcode)
  - Instruments (latest Xcode)
- HPCToolKit
  - From our friends at Rice University
  - mostly AMD and Intel processors and Linux OS
- CodeAnalyst
  - developed by AMD for profiling on Intel systems
  - Linux and Windows versions.
- Intel Vtune package
  - free versions are available for students
  - complicated to navigate their web pages…

Instrumenting

- At compile time, link time or at a later stage your binary code is altered to put calls into a timing library running inside your program
- Simplest is a compiler flag
  - g++ -pg
  - inserts gprof code at the entry and exit of every function
  - when your code runs it will generate a gmon.out file
  - >gprof a.out gmon.out >profile.txt
- Advantages
  - Full call graph, which accurate call counts
- Disadvantages
  - instrumentation has to be very lightweight or it will skew the results
  - can instrument at too fine a granularity
  - large functions might have too coarse a granularity.
  - doesn’t work on Apple computers.

a.out B.2, gprof a.out gmon.out

```
[1]     97.9    0.00    4.04                    main [1]
                0.00    4.04    200/200             femain(int, char**) [2]
-----------------------------------------------
                0.00    4.04    200/200             main [1]
[2]     97.9    0.00    4.04    200             femain(int, char**) [2]
                0.00    2.99    200/200           JacobiSolver::solve [3]
                0.02    0.81    200/200           FEPoissonOperator::FEPoissonOperator
                0.01    0.06    200/200           FEPoissonOperator::makeRHS [30]
                0.01    0.05    200/200           FEGrid::FEGrid(std::string const&, [32]
                0.00    0.03    200/200           reinsert(FEGrid const&… [38]
-----------------------------------------------
                0.00    2.99    200/200           femain(int, char**) [2]
[3]     72.6    0.00    2.99    200             JacobiSolver::solve(
                1.04    0.83  40400/40400         SparseMatrix::operator*
                0.14    0.16  40400/40400         operator+(std::vector<float>)
                0.21    0.07  40600/40600         norm(std::vector<float,….
```

You can notice that the resolution of gprof is pretty poor. things under 10ms are swept away

You can see that I put the main program inside it's own loop for 200 iterations of the whole solver.

Full Instrumentation used to make sense

- A function call used to be very expensive.
  - So, inserting extra code into the epilogue and prologue was low impact
- Special hardware in modern processors make most function calls about 40 times faster than 15 years ago.
- extra code in the epilogue prologue now seriously biases the thing being measured.
- Automatic full instrumentation is no longer in favor.

Manual Instrumentation

- An attempt to salvage the better elements of instrumentation
- Can be labor intensive
- Is also your only option for profiling parallel programs
- TAU is an example package
- For this course you will use one that we* wrote for you.

* “we” = Brian Van Straalen

CH_Timer manual profiling

```
-----------
Timer report 0 (46 timers)
--------------
-----------------------------------------------------------
[0]root 14.07030 1
    100.0% 14.0694        1 main [1]
    100.0%                  Total
-----------------------------------------------------------
[1]main 14.06938 1
    30.1%  4.2318       15 mg [2]
     2.6%  0.3675       16 resnorm [7]
    32.7%                  Total
-----------------------------------------------------------
[2]mg 4.23180 15
    100.0%  4.2318       15 vcycle [3]
    100.0%                  Total
-----------------------------------------------------------
[3]vcycle 4.23177 15
    62.3%  2.6354       30 relax [4]
    25.9%  1.0965       15 vcycle [5]
     3.0%  0.1282       15 avgdown [10]
     3.0%  0.1276       15 fineInterp [11]
    94.2%                  Total
-----------------------------------------------------------
```

Using CH_Timer

```
#include "CH_Timer.H"

MultigridClass::vcycle(...)
{
  CH_TIMER("vcycle"); // times everything to end of
  scope.
  CH_TIMERS("vcycle phase 1", t1);
  CH_TIMERS("vcycle phase 2", t2);
  ...
  CH_START(t1);
   ...
  CH_STOP(t1);
  ...
  CH_START(t2);
   ...
  CH_STOP(t2);
}
```

CH_Timer.H defines several timing *macros*. These macros create objects on the stack which start the timer when they are constructed, and stops their timer when they go out of scope.

updates to your GNUmakefile

```
CT=$(HOME)/timer

SRC += CH_Timer.cpp

CFLAGS = -O3 -Wall -I$(CT) –L$(CT)

...
```

- OK, sampling profiler, gprof, your own tools, or our instrumenting timer system. Some tools at your disposal
- You've seen your first tool, compiler flags.
  - be wary of extremely high optimization levels O4, O5 etc.
- Let's measure some of the homework assignments and see what we can learn.

FFT3D (256^3) using BRI.

```
-----------------------------------------------------------
[0]root 11.23750 1
    32.7%  3.6745       1 fftmdForward [1]
    32.7%  3.6700       1 fftmdInverse [2]
    65.4%                 Total
-----------------------------------------------------------
[1]fftmdForward 3.67454 1
    67.6%  2.4829  196608 forward1d [3]
    95.7%                 Total
-----------------------------------------------------------
[2]fftmdInverse 3.66996 1
    67.6%  2.4811  196608 reverse1d [4]
    67.6%                 Total
-----------------------------------------------------------
```

- We actually can compute the total number of Flops: 256^3x8x3x5 = 2x10^9. Flop rate = 590 Mflops.
- But there is a large chunk of time not accounted for.

FFT3D (256^3) using BRI.

```
for (int dir = 0;dir < DIM; dir++)
    {
     ...
      Point edir = getUnitv(dir);
      for (Point pt=base.getLowCorner();base.notDone(pt);base.increment(pt))
        {

          for (int l = 0 ; l < m_N/2;l++)
            {
              f1d[l] = a_f[pt + edir*l]; // Copying from 3D array to 1D array.
              f1d[m_N - l-1] = a_f[pt-edir*(l+1)];
            }
          CH_START(t1);
          m_fft1dPtr->forwardFFTCC(fHat1d,f1d);
          CH_STOP(t1);

          for (int l = 0 ; l < m_N/2;l++)
            {
              a_f[pt + edir*l] = fHat1d[l];
              a_f[pt-edir*(l+1)] = fHat1d[m_N-l-1];
            }
        }
     }
```

- Is copying the culprit?
- Unit stride or non-unit-stride ?
- Add additional timer for copy loops at dir != 0;

FFT3D (256^3) using BRI.

```
----------------------------------------------------------
[0]root 11.23750 1
    32.7%  3.6745        1 fftmdForward [1]
    32.7%  3.6700        1 fftmdInverse [2]
    65.4%                  Total
----------------------------------------------------------
[1]fftmdForward 3.67454 1
    67.6%  2.4829   196608 forward1d [3]
    16.8%  0.6183   131072 copyin    [6]
    11.3%  0.4165   131072 copyout   [7]
    95.7%                  Total
----------------------------------------------------------
[2]fftmdInverse 3.66996 1
    67.6%  2.4811   196608 reverse1d [4]
    67.6%                  Total
----------------------------------------------------------
```

- dir != 0 copying = 1.03 secs (we infer that unit stride is the remaining .18 secs).
- Flop rate for 1D FFTs is $2x10^9/2.49$ = 800 Mflops.
- Other measurements indicate that FFTW is about twice as fast as FFTBRI, so we expect that the flop rate here should double, to 1600 Mflops. However, the net flop rate with FFTW and copying is $2x10^9/(1.25 + 1.2)$ = 890 MFlops (as opposed to 590 Mflops).
- Tuned multidimensional FFTs? Make our copying faster ?

Unstructured Grid

```
>./fesolver.exe ../FEData/B.2
number rows, nonzeros = 279 , 1809
Intial RHS norm 0.482459
the number of iterations = 601
Final Solver residual was 9.9861e-07
```

- Total number of flops in Jacobi = 601x1809x2 = 2.17x10^6

```
---------------------------------------------------------
[1]main 0.01236 1
    31.0%  0.0038        1 Jacobi [2]
    31.0%                  Total
---------------------------------------------------------
[2]Jacobi 0.00383 1
```

- Flop rate = 2.17x10^6/.00383 = 567 Mflops. I would like to run this on a large problem before I completely believe this number. However, it is ok for a start (average row length is 6; indirect addressing in accessing the vector in matrix-vector multiply).

Structured Grid – Point Jacobi.

```
CH_TIMER("loop",t1);
...
CH_START(t1);
  for (Point p=D0.getLowCorner();D0.notDone(p);D0.increment(p))
    {
      LOfPhi[p] = 0.;
      for (int dir = 0;dir < DIM; dir++)
        {
          LOfPhi[p] += phi[p+getUnitv(dir)]
            + phi[p-getUnitv(dir)];
        }
      LOfPhi[p] -=2*DIM*phi[p];
      LOfPhi[p] *= hsqinv;
      LOfPhi[p] -= f[p];
      maxL = max(abs(LOfPhi[p]),maxL);
    }
  if (k == 0 ) maxL0 = maxL;
  for (Point p=D0.getLowCorner();D0.notDone(p);D0.increment(p))
    {
      phi[p] += lambda*(LOfPhi[p]);
    }
  CH_STOP(t1);
```

Structured Grid – Point Jacobi.

```
 > ./mdArrayTest2D.exe
input size in each direction (2 ints)
64 64
number of iterations
2000
max norm of residual = 0.23128, iteration = 2000
3.05176e-05 = relaxation parameter
```

- Number of flops = 64^2x10x2000 = 8.2x10^7 flops.

```
---------------------------------------------------------
[0]root 4.73420 1
     6.2%  0.2916        1 main [1]
     6.2%                  Total
---------------------------------------------------------
[1]main 0.29160 1
    98.7%  0.2879     2000 loop [2]
    98.7%                  Total
---------------------------------------------------------
[2]loop 0.28786 2000[1]
```

- Flop rate = 285 Mflops. Seems low, given that stencil operations are unit stride. And I've been doing some optimization already.

Naïve vs. Vendor DGEMM Bounds Expectations

```
>./naive.exe
n 31, MFlop/sec = 2018.29
n 32, MFlop/sec = 1754.92
n 96, MFlop/sec = 1746.74
n 97, MFlop/sec = 1906.88
n 127, MFlop/sec = 1871.38
n 128, MFlop/sec = 1674.05
n 129, MFlop/sec = 1951.06
n 191, MFlop/sec = 1673.44
n 192, MFlop/sec = 1514.24
n 229, MFlop/sec = 1915.5
n 255, MFlop/sec = 1692.96
n 256, MFlop/sec = 827.36
n 257, MFlop/sec = 1751.56
n 319, MFlop/sec = 1762.5
n 320, MFlop/sec = 1431.29
n 321, MFlop/sec = 1714.46
n 479, MFlop/sec = 1569.42
n 480, MFlop/sec = 1325.46
n 511, MFlop/sec = 1242.37
n 512, MFlop/sec = 645.815
n 639, MFlop/sec = 247.698
n 640, MFlop/sec = 231.998
n 767, MFlop/sec = 211.702
n 768, MFlop/sec = 221.34
n 769, MFlop/sec = 204.241
```

```
>./blas.exe
n 31, MFlop/sec = 8828.4
n 32, MFlop/sec = 11479.1
n 96, MFlop/sec = 17448.5
n 97, MFlop/sec = 14472.2
n 127, MFlop/sec = 15743.9
n 128, MFlop/sec = 16956.6
n 129, MFlop/sec = 19335.8
n 191, MFlop/sec = 25332.7
n 192, MFlop/sec = 26786
n 229, MFlop/sec = 27853.2
n 255, MFlop/sec = 28101
n 256, MFlop/sec = 30022.1
n 257, MFlop/sec = 28344.9
n 319, MFlop/sec = 28477
n 320, MFlop/sec = 28783.5
n 321, MFlop/sec = 28163.6
n 479, MFlop/sec = 29673.5
n 480, MFlop/sec = 30142.8
n 511, MFlop/sec = 29283.7
n 512, MFlop/sec = 30681.8
n 639, MFlop/sec = 28603.6
n 640, MFlop/sec = 31517.6
n 767, MFlop/sec = 29292.7
n 768, MFlop/sec = 31737.5
n 769, MFlop/sec = 29681.4
```

Inlining

- Function calls are faster than in the bad old days, but still not free
  - Every function call inserts a jmp instruction in the binary code
  - arguments are copied
  - compilers today still do not optimize instruction scheduling across function calls.
    - Out-of-order processors *try* to do this, but have limited smarts
  - function calls in your inner loops should be avoided
- But functions let me create maintainable code
  - we can write operator[] once and debug it and rely on it
  - we encapsulate the implementation from the user
    - freeing us to alter the implementation when we need to
- *inlining* is a way of telling the compiler to not really create a function, just the function *semantics*.

Inlining cont. Recall from Homework1

```
for (Point pt=domain.getLowCorner();
      domain.notDone(pt);domain.increment(pt));
   {
     LOfPhi[pt] = 0.;
     for (int dir = 0;dir < DIM; dir++)
       {
         Point edir=getUnitv(dir);
         LOfPhi[pt] += phi[pt + edir] + phi[pt-edir]
       }
      LOfPhi[pt] -=2*DIM*phi[pt];
   }
```

This is a BIG loop. we'll execute the code here O(N) times

Look at all the function calls

Inlining cont.

- We would like the compiler to be smarter and just insert the body of these inner-loop functions right into the place where the compiler can schedule all the operations together.
- This takes two steps:

1. The *declaration* needs to declare this function should be inlined
2. You need to provide the inlined *definition* in the header file

- This means the definition is not in the source (.cpp) file now.
- General rule for inline functions: When the function body is probably less cost than invoking the function itself *and* is likely to be invoked in with O(N) code.

Inlining demonstrated

```
inline T& operator[](const Point& a_iv)
{
  int k = m_box.getIndex(a_iv);
  return m_rawPtr[k];
};
...
inline unsigned int DBox::getIndex(const Point& a_pt) const
{
  unsigned int factor = 1;
  unsigned int linIndex = a_pt[0] - m_lowCorner[0];
  for (unsigned char i = 1;i < DIM;i++)
    {
      factor = factor*(m_highCorner[i-1] - m_lowCorner[i-1]+1);
      linIndex = linIndex + (a_pt[i] - m_lowCorner[i])*factor;
    }
  return linIndex;
}
```

Even with inlining, we are still low.

- Inlining is only advice to the compiler – not compulsory.
- Multiple levels of indirection may be hiding the essential stride-one access.

You will explore some of these possibilities in homework 5.

Lift loop invariants out of loops

```
for (int i=0;i<domainWithGhost.sizeOf();i++)
    {
      Point pt = domainWithGhost.getIndex(i);
      double val = 1.;
      for (int dir = 0;dir < DIM; dir++)
        {
          val *=    sin(2*M_PI*pt[dir]*h);
        }
      phi[i] = val;
    }
```

- The compiler should be smart enough to do this for you, so don’t be surprised when your -O3 runs the same speed….but your debug code *will* run faster

Loop Fusion

```
float maxD=FLT_MIN, minD=FLT_MAX, sumSquareD=0;
for(unsigned int i=0; i<d.size(); i++)
  {
    sumSquareD += d[i]*d[i];
  }
for(unsigned int i=0; i<d.size(); i++)
  {
    maxD = std::max(d[i],maxD);
  }
for(unsigned int i=0; i<d.size(); i++)
  {
    minD = std::min(d[i], minD);
  }
```

Or with these loops fused together (plus telling the compiler to access the vector only once).

```
for(unsigned int i=0; i<d.size(); i++)
  {
    float x = d[i];
    sumSquareD += x*x;
    maxD = std::max(x,maxD);
    minD = std::min(x,minD);
  }
```

Standard Template Library helpers

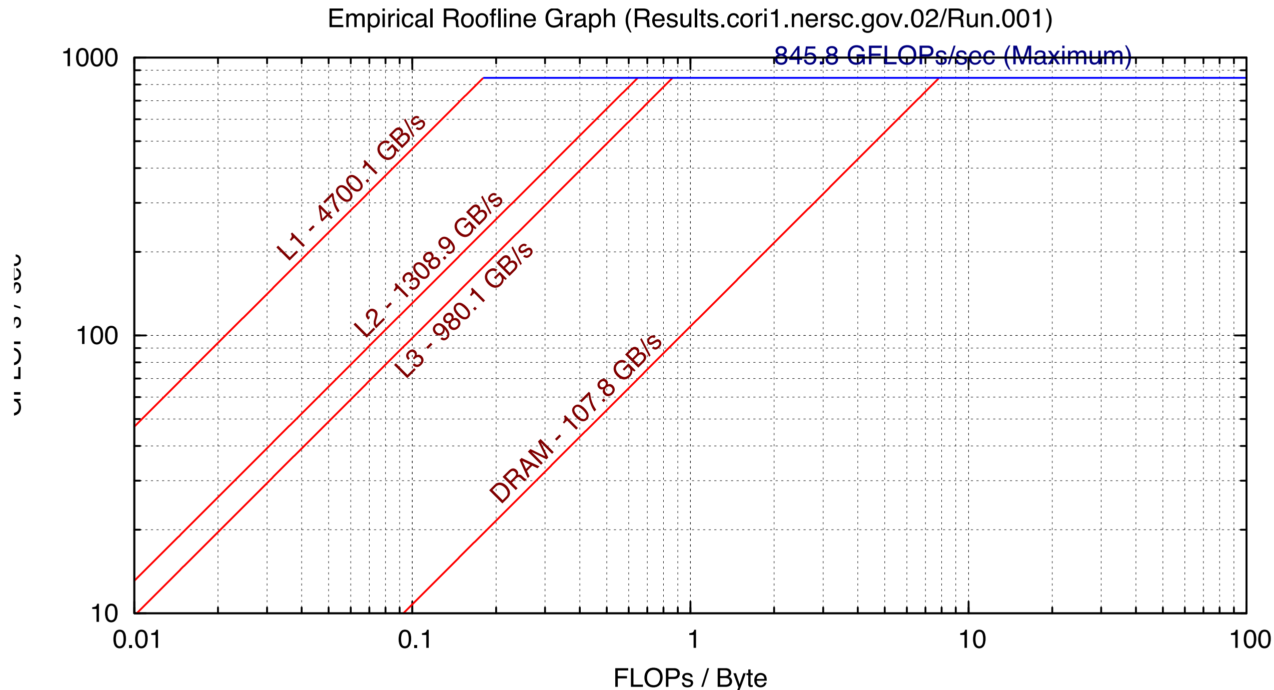
- std::swap (<algorithms>)
  - Many STL containers can quickly swap their contents with another of their own type.
  - vector a, b; a.swap(b); std::swap(a,b);
- std::shared_ptr (<memory>)
  - When you have several objects that need to share access to a particular class instantiation. Keeps you from having to make multiple copies, but safely (i.e. without memory leaks).

More esoteric optimizations

- We don't tend to teach the higher order optimizations past these the ones given
  - Loop unrolling
  - Loop strip-mining
  - Loop pipelining
  - SIMD instructions
- With templates (remember, templates get evaluated in the early compiler stage), and inlining, compilers are pretty good about doing these thing for you
- But they are poor at optimizing across function boundaries (at least, for now…that's an active area of research)
  - They also cannot optimize across source files.
- These are all diminishing returns
  - The biggest wins for large simulations is your choice of
    - algorithm
    - data structures

Back to Performance Models

- Our model was “naïve degemm should predict what to expect”.
- There are better models – roofline is one.



- Issues
  - You have to be able to perform the measurments.
  - There are hidden parameters to this model, e.g. working set size.