CS 294-73
Software Engineering for Scientific Computing

Lecture 14: 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 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 / word
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 / word (“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
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.
  - Tradeoffs: I am willing to give up 2x for easily maintained and modified code, but not 10x.
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
PR_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 Proto Timers

#include "Proto_Timer.H"

MultigridClass::vcycle(...)
{
    PR_TIMER("vcycle"); // times everything to end of scope.
    PR_TIMERS("vcycle phase 1", t1);
    PR_TIMERS("vcycle phase 2", t2);
    ...
    PR_START(t1);
    ...
    PR_STOP(t1);
    ...
    PR_START(t2);
    ...
    PR_STOP(t2);
}

Proto_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.
Structured Grid – Point Jacobi.

```cpp
{PR_TIME("Stencil Evaluation");
 for (int iter = 0; iter < 100; iter++)
 {
 for (auto it = D0.begin(); !it.done(); ++it)
 {
     Point p = *it;
     LOfPhi(p) = 0.;
     for (int dir = 0; dir < DIM; dir++)
     {
         LOfPhi(p) += phi(p+e[dir])
         + phi(p-e[dir]);
     }
     LOfPhi(p) -= 2*DIM*phi(p);
     LOfPhi(p) *= hsqi;
     LOfPhi(p) -= f(p);
 }
 for (auto it = D0.begin(); !it.done(); ++it)
 {
     Point p = *it;
     phi(p) += lambda*(LOfPhi(p));
 }
 }
}
```
Structured Grid Operator Evaluation.

> ./mdArrayTest2D.exe

[0]root 0.07139 1
  98.6%  0.0704  1 Stencil Evaluation [1]
  0.0%  0.0000  1 BoxData::setval [2]
  0.0%  0.0000  3 BoxData::define(Box) (memory allocation) [3]
  98.6%  0.0000  Total

---------------------------------------------------------
[1]Stencil Evaluation 0.07041 1

---------------------------------------------------------
[2]BoxData::setval 0.00001 1
  1.4%  0.0000  1 slice(BoxData<T,C,D,E>&, int, int, int) [4]
  1.4%  Total

---------------------------------------------------------
[3]BoxData::define(Box) (memory allocation) 0.00000 3

---------------------------------------------------------
[4]slice(BoxData<T,C,D,E>&, int, int, int) 0.00000 1

• Flop rate = 46 Mflops, compared to 2 Gflops for triply-nested-loop DGEMM.
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.

• 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 is advice to the compiler – it will make a decision on whether it is actually worthwhile.
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 is advice to the compiler – it will make a decision on whether it is actually worthwhile.
- All of the indexing operations for BoxData are inlined.
In Proto_BoxData.H

inline T& operator()(const Point& a_pt,
                     unsigned int  a_c = 0,
                     unsigned char a_d = 0,
                     unsigned char a_e = 0)
{
    ...
    return m_rawPtr[index(a_pt,a_c,a_d,a_e)];
}
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 only dereference the vector 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).
What drives the differences in performance?

• Answer: Failure to take advantage of locality.
  - Reuse of data in higher levels of cache.
  - Failure to take advantage of hardware features e.g. SIMD.
Multigrid

\[ vcycle(\phi, \rho) \]
\[
\begin{cases}
\phi := \phi + \lambda(L(\phi) - \rho) \text{ } p \text{ } \text{times} \\
\text{if (level > 0)} \\
\{ \\
\mathcal{R} = \rho - L(\phi) \\
\mathcal{R}_c = \mathcal{A}(\mathcal{R}) \\
\delta : B_c \rightarrow \mathbb{R}, \delta = 0 \\
vcycle(\delta, \mathcal{R}_c) \\
\phi := \phi + \mathcal{I}(\delta) \\
\phi := \phi + \lambda * (L(\phi) - \rho) \text{ } p \text{ } \text{times} \\
\} \\
\text{else} \\
\{ \\
\phi := \phi + \lambda * (L(\phi) - \rho) \text{ } p_B \text{ } \text{times} \\
\} \\
\end{cases}
\]

At the top level, iterate until residual is reduced by some large factor.
Multigrid v-cycle.

Multigrid::vCycle(...) 
{
...  
if (m_level > 0)  
{
    pointRelax(a_phi,a_rhs,m_preRelax);
    residual(m_res,a_phi,a_rhs);
    avgDown(m_resc,m_res);
    m_delta.setVal(0.);
    m_coarsePtr->vCycle(m_delta,m_resc);
    fineInterp(a_phi,m_delta);
    pointRelax(a_phi,a_rhs,m_postRelax);
}
else
pointRelax(a_phi,a_rhs,m_bottomRelax);
}
Baseline implementation of Residual

Multigrid::residual(...)
{
...
res.setVal(0.);
for (auto it = bx.begin(); !it.done(); ++it)
{
    Point pt = *pt;
    for (int dir = 0; dir < DIM; dir++)
    {
        res(pt) += (a_phi(pt + e[dir]) + a_phi(pt - e[dir]));
    }
    res(pt) -= -2*DIM*a_phi(pt)
    res(pt) = res(pt)*hsqi - a_rhs(pt);
}
}
Idea: grab contiguous chunks of data, and work on each of them.

```c
int high = ...
for (int k = 0; k < high; k++)
{
    a[k] = b[k] * c[k] + d[2*k];
}
```

Compiler is smart enough to recognize this data access pattern, and optimize it. But how do we get this into our stencil code?
Pencil implementation of Residual

Multigrid::residual(...)  
{
  ...  
  res.setVal(0.);
  Box bxbase = bx.lowEdge(0);
  for (auto it = bxBase.begin(); !it.done(); ++it)
  {
    Point ptBase = *it;
    double& resptr = &res(ptBase);
    double& rhsptr = &a_rhs(ptBase);
    double& phiptr = &a_phi[ptBase];
    for (int dir = 0; dir < DIM; ++dir)
    {
      double& phiptrp = phiptr(pt+e(dir));
      double& phiptrm = phiptr(pt-e(dir));
      for (int k = 0; k < length; ++k)
      {
        resptr[k] += phiptrp[k] + phiptrm[k];
      }
      for (int k = 0; k < length; ++k)
      {
        resptr[k] -= -2*DIM*phiptr[k]
        resptr[k] = resptr[k]*hsqi - rhsptr[k];
      }
    }
}
Pencils only work if unambiguous

double* a, b, c, d;
...
for (int k = 0; k < bar.size(); k++)
{
    a[k] = b[k] * c[k] + d[2*k];
}
Fails. Instead, use
double* a, b, c;
...
int high = bar.size();
for (int k = 0; k < high; k++)
{
    a[k] = b[k] * c[k] + d[2*k];
}

Loop bodies have to have obvious constant-stride access, fixed loop limits. If we replace double* a, b, c, d; with vector<double> a, b, c, d; will the compiler recognize the unit stride access?
Pipelining and SIMD
What is Pipelining?

Dave Patterson's Laundry example: 4 people doing laundry
wash (30 min) + dry (40 min) + fold (20 min) = 90 min

Latency

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 PM</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

- In this example:
  - Sequential execution takes \(4 \times 90 \text{ min} = 6 \text{ hours}\)
  - Pipelined execution takes \(30 + 4 \times 40 + 20 = 3.5 \text{ hours}\)

- **Bandwidth** = loads/hour
  - \(\text{BW} = 4/6 \text{ l/h} \) w/o pipelining
  - \(\text{BW} = 4/3.5 \text{ l/h} \) w pipelining
  - \(\text{BW} \leq 1.5 \text{ l/h} \) w pipelining, more total loads

- Pipelining helps bandwidth but not latency (90 min)
- Bandwidth limited by slowest pipeline stage
- Potential speedup = Number pipe stages

A
B
C
D

- 6 PM
- 7
- 8
- 9

Potential speedup = Number pipe stages
Example: 5 Steps of MIPS Datapath

Figure 3.4, Page 134, CA:AQA 2e by Patterson and Hennessy

- Pipelining is also used within arithmetic units
  - a fp multiply may have latency 10 cycles, but throughput of 1/cycle
SIMD: Single Instruction, Multiple Data

- **Scalar processing**
  - traditional mode
  - one operation produces one result

- **SIMD processing**
  - with SSE / SSE2
  - SSE = streaming SIMD extensions
  - one operation produces multiple results

Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation
SSE / SSE2 SIMD on Intel

- SSE2 data types: anything that fits into 16 bytes, e.g.,

- Instructions perform add, multiply etc. on all the data in this 16-byte register in parallel

- Challenges:
  - Need to be contiguous in memory and aligned
  - Some instructions to move data around from one part of register to another
  - Similar on GPUs, vector processors (but many more simultaneous operations)
Vectorization is done by the compiler (mostly).

Vectorization
Using clang++:
CXXFLAGS += -Rpass=loop-vectorize
If you want to know what loops are failing to vectorize, and why,
CXXFLAGS += -Rpass=analysis=loop-vectorize

Using g++:
CXXFLAGS+=-ftree-vectorizer verbose=2
To turn off vectorization:
CXXFLAGS += -fno-vectorize
(However, g++ is more pessimistic about the value of vectorization).
An Embedded Domain-Specific Language

At a high level, code looks like our pseudocode, but inside it is messy, particularly when you start optimizing for performance.

Can we do better?
• Identify programming abstractions as aggregate operations corresponding to the high-level mathematical abstractions.
• Implement as C++ classes that contain the optimizations you would otherwise do by hand. The idea is that the opportunities for obtaining high performance come from the mathematical structure of the algorithms.
• (apply compiler technologies to make it even better).

Proto does half of this, i.e. defines high-level data structures. We’ll now take a look at the other half of the Proto infrastructure, which is a stencil language.

(Joint work with Brian Van Straalen, Dan Graves, Chris Gebhart).
**Stencil Operators**

\[ \phi : B \rightarrow \mathbb{R}^N, \text{ } B \text{ a Box} \]

\[ L(\phi)_i = \sum_{s \in S} a_s \phi_{i+s}, \text{ } i \in B' \]

Can think of the operator \( L \) as an object that acts any BoxData.

\[ L = \sum_{s \in S} a_s S^s, \text{ } S^s (\phi)_i = \phi_{i+s} \]

Stencil operators have their own algebra: you can add them, compose them, multiply them by scalars, without knowing the details of what they will be applied to.

\[ L_{1,2} = \sum_{s \in S} a_s^{1,2} S^s \]

\[ L_1 \circ L_2 = \sum_t \left( \sum_{s+s' = t} a_s^1 a_s'^2 \right) S^t \]
Stencil Operators

Stencil<double> m_Lap2nd;

m_Lap2nd =
    (-2.0*DIM)*Shift(getZeros());

for (int dir = 0; dir < DIM ; dir++)
{
    Point edir = Point::Basis(dir);
    Stencil<double> plus = 1.0*Shift(edir);
    Stencil<double> minus = 1.0*Shift(edir*(-1));

    m_Lap2nd = m_Lap2nd + minus + plus;
}
Stencil Operators

Stencil<double> m_Lap2nd;

m_Lap2nd =

(-2.0*DIM)*Shift(getZeros());

for (int dir = 0; dir < DIM ; dir++)
{
    Point edir = getUnitv(dir);
    Stencil<double> plus = 1.0*Shift(edir);
    Stencil<double> minus = 1.0*Shift(edir*(-1));

    m_Lap2nd = m_Lap2nd + minus + plus;
}
Multigrid::residual(
    BoxData<double> & a_res,
    BoxData<double> & a_phi,
    BoxData<double> & a_rhs
)
{
    PROTO_TIMERS("residual");
    getGhost(a_phi);
    double hsqi = 1.0/(m_dx*m_dx);
    a_res |= m_Lap2nd(a_phi, hsqi);
    a_res += a_rhs;
};

Apply the operator and store in the lhs.
Stencils can be strided, both on input and on output. Useful for coarsening / refinement.

\[ \mathcal{L}(\phi)_{ir^{\text{dest}}+q^{\text{dest}}} = \sum_s a_s \phi_{ir^{\text{src}}+s}, i \in \Gamma \]
Arithmetic on BoxData

... 
    a_res += a_rhs;
...

Defined on intersection of the two Boxes. In fact, all of the stencil operations come with Box inference.

Pointwise operators.

\[(f@U)_i \equiv f(U_i), i \in B\]

forall(fOfU,U,f,bx); // f = f(U_in, U_out).

we can also use use lambda calculus in C++11 to apply partially evaluated functions.
forall(fOfU,U,[dt](State& a, const State& b){return f(a,b,dt);}), bx);

Use lambdas to apply member functions of a class.
forall(W, U, 
    [this](State& a, const State& b){return consToPrim(a,b);}, B_3);
How do we get performance?

In the apply function, we implement the pencil construction once.

... 
for (int isten =0;isten < srcOffset.size(); isten++)
{
    int kpoffset = kbasep+srcOffset[isten];
    double coefpt = coef[isten];
    for (int k0 = 0;k0 < nptsDst;k0++)
    {
        const T& phival = (phi_ptr_i)[kpoffset+k0];
        T& lofphi = (lofphi_ptr_i)[kbasel+k0];
        lofphi+=coefpt*phival;
    }
}
...

Multigrid in a slide

// residual.
a_res = m_Lap2nd(a_phi, -hsqi);
a_res += a_rhs;

// pointRelax:
for (int iter = 0; iter < a_numIter; iter++)
{
    getGhost(a_phi);
    {lofphi = Identity(a_rhs, -m_lambda);
        lofphi += m_Jacobi(a_phi);
        lofphi.copyTo(a_phi);}
}

// fineInterp:
for (auto it = m_boxSten.begin(); !it.done(); ++it)
{a_phi += m_Interp(ptsten)(a_delta);}

// avgdown:
a_resc = m_avgDown(a_res);