CS 294-73
Software Engineering for Scientific Computing

Lecture 14: Performance on cache-based systems, Profiling & Tips for C++

Guest Lecture: Brian Van Straalen
Some Slides from James Demmel and Kathy Yelick
Motivation

• Most applications run at < 10% of the “peak” performance of a system
  • Peak is the maximum the hardware can physically execute
• Much of this performance is lost on a single processor, i.e., the code running on one processor often runs at only 10-20% of the processor peak
• Most of the single processor performance loss is in the memory system
  • Moving data takes much longer than arithmetic and logic

• To understand this, we need to look under the hood of modern processors
  • For today, we will look at only a single “core” processor
  • These issues will exist on processors within any parallel computer
Outline

• Memory hierarchies
  • Temporal and spatial locality
  • Basics of caches
  • Use of microbenchmarks to characterized performance

• Case study: Matrix Multiplication
• Roofline Model
• Profiling
• Random Tips for C++
Approaches to Handling Memory Latency

- Bandwidth has improved more than latency
  - 23% per year vs 7% per year
- Approach to address the memory latency problem
  - Eliminate memory operations by saving values in small, fast memory (cache) and reusing them
    - need temporal locality in program
  - Take advantage of better bandwidth by getting a chunk of memory and saving it in small fast memory (cache) and using whole chunk
    - need spatial locality in program
  - Take advantage of better bandwidth by allowing processor to issue multiple reads to the memory system at once
    - concurrency in the instruction stream, e.g. load whole array, as in vector processors; or prefetching
  - Overlap computation & memory operations
    - prefetching
Programs with locality cache well ...

Memory Hierarchy

• Take advantage of the principle of locality to:
  • Present as much memory as in the cheapest technology
  • Provide access at speed offered by the fastest technology

---

Latency (ns): ~1  
Size (bytes):  
- Core cache: ~10^6  
- Shared Cache: O(10^6)  
- Second Level Cache (SRAM): ~5-10  
- Main Memory (DRAM/FLASH/PCM): ~100  
- Secondary Storage (Disk/FLASH/PCM): ~10^7  
- Tertiary Storage (Tape/Cloud Storage): ~10^10

---
Cache Basics

- **Cache** is fast (expensive) memory which keeps copy of data in main memory; it is hidden from software
  - Simplest example: data at memory address xxxxx1101 is stored at cache location 1101
- **Cache hit**: in-cache memory access—cheap
- **Cache miss**: non-cached memory access—expensive
  - Need to access next, slower level of cache
- **Cache line length**: # of bytes loaded together in one entry
  - Ex: If either xxxxx1100 or xxxxx1101 is loaded, both are
- **Associativity**
  - Direct-mapped: only 1 address (line) in a given range in cache
    - Data stored at address xxxxx1101 stored at cache location 1101, in 16 word cache
  - n-way: \( n \geq 2 \) lines with different addresses can be stored
    - Example (2-way): addresses xxxxx1100 can be stored at cache location 1101 or 1100.
Why Have Multiple Levels of Cache?

• On-chip vs. off-chip
  • On-chip caches are faster, but limited in size
• A large cache has delays
  • Hardware to check longer addresses in cache takes more time
  • Associativity, which gives a more general set of data in cache, also takes more time

• Some examples:
  • Cray T3E eliminated one cache to speed up misses
  • IBM uses a level of cache as a “victim cache” which is cheaper
• There are other levels of the memory hierarchy
  • Register, pages (TLB, virtual memory), … (Page (memory))
  • And it isn’t always a hierarchy
In practice, what does memory access look like

Streaming Triad Benchmark

```c
for(int i=0; i<WorkingSetSize/(3*sizeof(a[0])); i++)
{
    c[i] = f(a[i], b[i]);
}
```
What if I do lots of operations for each access?
Lessons

• Actual performance of a simple program can be a complicated function of the architecture
  • Slight changes in the architecture or program change the performance significantly
  • To write fast programs, need to consider architecture
  • We would like simple models to help us design efficient algorithms

• We will illustrate with a common technique for improving cache performance, called blocking or tiling
  • Idea: used divide-and-conquer to define a problem that fits in register/L1-cache/L2-cache
Outline

• Memory hierarchies
  • Use of microbenchmarks to characterized performance

• Case study: Matrix Multiplication
  • Use of performance models to understand performance
  • Simple cache model
  • Warm-up: Matrix-vector multiplication
  • (continued next time)

• Roofline Model

• Profiling

• Random C++ Tips
Why Matrix Multiplication?

• An important kernel in many problems
  • Appears in many linear algebra algorithms
    • Bottleneck for dense linear algebra
  • One of the 7 motifs
    • Closely related to other algorithms, e.g., transitive closure on a graph using Floyd-Warshall

• Optimization ideas can be used in other problems
• The best case for optimization payoffs
• The most-studied algorithm in high performance computing
What do commercial and CSE applications have in common?

Motif/Dwarf: Common Computational Methods (Red Hot → Blue Cool)

<table>
<thead>
<tr>
<th></th>
<th>Embed</th>
<th>SPEC</th>
<th>DB</th>
<th>Games</th>
<th>ML</th>
<th>HPC</th>
<th>Health</th>
<th>Image</th>
<th>Speech</th>
<th>Music</th>
<th>Browser</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Finite State Mach.</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>2</td>
<td>Combinational</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>3</td>
<td>Graph Traversal</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>4</td>
<td>Structured Grid</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>5</td>
<td>Dense Matrix</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>6</td>
<td>Sparse Matrix</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>7</td>
<td>Spectral (FFT)</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>8</td>
<td>Dynamic Prog</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>9</td>
<td>N-Body</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>10</td>
<td>MapReduce</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>11</td>
<td>Backtrack/ B&amp;B</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>12</td>
<td>Graphical Models</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>13</td>
<td>Unstructured Grid</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>
Matrix-multiply, optimized several ways

Speed of n-by-n matrix multiply on Sun Ultra-1/170, peak = 330 MFlops
Naïve Matrix Multiply on RS/6000

![Graph showing the relationship between log cycles/flop and log problem size. The graph indicates that there is a page miss every iteration, a TLB miss every iteration, and a cache miss every 16 iterations and every 512 iterations.](image-url)
Outline

• Idealized and actual costs in modern processors
• Case Study: Matrix Multiplications
• Roofline Model
  • A simple model that allows us to understand algorithmic tradeoffs
Roelfine Performance Model

• Kernels are characterized
  • Calculations they perform: \( W \) (“W”ork)
  • Data they read/write: \( Q \)
  • Arithmetic Intensity: \( A = \frac{W}{Q} \) (ie flops/bytes)

• Data re-use distance
  • Working set size (bytes)

```plaintext
(a) temp=0.0;
for(i=0;i<N;i++){
    temp = A[i]*A[i];
}
magnitude = sqrt(temp);

(b) for(i=0;i<N;i++){
    for(j=0;j<N;j++){
        C[i,j] = a*A[i,j] +
               b*( A[i,j-1] + A[i-1,j] +
                    A[i+1,j] + A[i,j+1] );
    }
}

(c) C[i,j]=0.0;
for(i=0;i<N;i++){
    for(j=0;j<N;j++){
        for(k=0;k<N;k++){
            C[i,j] += A[i,k]*B[k,j];
        }
    }
}
```
Roofline figure for Intel Haswell
Loop Fusion

```c
float maxD=FLT_MIN, minD=FLT_MAX, sumSquareD=0;
for(unsigned int i=0; i<d.size(); i++)   // AI=0.25
{
    sumSquareD += d[i]*d[i];
}
for(unsigned int i=0; i<d.size(); i++)  // AI=.125
{
    maxD = std::max(d[i],maxD);
}
for(unsigned int i=0; i<d.size(); i++)  //AI= 0.125
{
    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++)   //AI= 0.5
{
    float x = d[i];
    sumSquareD += x*x;
    maxD = std::max(x,maxD);
    minD = std::min(x,minD);
}
```
Step one 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
  • Do 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: 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.
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 20 years ago.
• extra code in the epilogue prologue now seriously biases the thing being measured.
• Automatic full instrumentation is no longer in favor.
• Compiler-based instrumentation tends to over-instrument
  • small short functions get heavily skewed by the measuring process
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 I wrote for you.
TraceTimer manual profiling

[0] main 0.01370 100.0% 1
  [1] FEGrid::FEGrid(....) 0.00374 27.3% 1
  [2] JacobiSolver::solve 0.00331 24.2% 1
    [6] SparseMatrix::operator* 0.00216 15.8% 202
  [9] vector<float> operator- 0.00026 1.9% 202
 [10] norm 0.00024 1.8% 203
 [11] vector<float> operator+ 0.00023 1.7% 202
 [12] vector<float> operator* 0.00018 1.3% 202
   [17] SparseMatrix::operator[] const 0.00001 0.1% 279
  [3] FEPoissonOperator::FEPoissonOperator(..) 0.00286 20.8% 1
  [5] buildIntegral 0.00250 18.3% 4714
    [7] FEGrid::gradient 0.00080 5.8% 9428
    [8] SparseMatrix::operator[] 0.00056 4.1% 4714
  [4] FEWrite 0.00277 20.2% 1
  [13] FEPoissonOperator::makeRHS 0.00007 0.5% 1
  [14] SparseMatrix::symmetric() 0.00007 0.5% 1
    [16] SparseMatrix::operator[] const 0.00003 0.2% 765
  [15] FEGrid::centroid 0.00004 0.3% 611
  [18] SparseMatrix::operator[] const 0.00001 0.1% 279
<table>
<thead>
<tr>
<th>Function</th>
<th>Time</th>
<th>Calls</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]main</td>
<td>0.01370</td>
<td>1</td>
<td>FEGrid::FEGrid(....) [1]</td>
</tr>
<tr>
<td></td>
<td>27.3%</td>
<td>0.0037</td>
<td>1 FEGrid::FEGrid(....) [1]</td>
</tr>
<tr>
<td></td>
<td>24.2%</td>
<td>0.0033</td>
<td>1 JacobiSolver::solve [2]</td>
</tr>
<tr>
<td></td>
<td>20.8%</td>
<td>0.0029</td>
<td>1 FEpoissonOperator::FEpoissonOperator(..) [3]</td>
</tr>
<tr>
<td></td>
<td>20.2%</td>
<td>0.0028</td>
<td>1 FEWrite [4]</td>
</tr>
<tr>
<td></td>
<td>0.5%</td>
<td>0.0001</td>
<td>1 FEpoissonOperator::makeRHS [13]</td>
</tr>
<tr>
<td></td>
<td>0.5%</td>
<td>0.0001</td>
<td>1 SparseMatrix::symmetric() [14]</td>
</tr>
<tr>
<td></td>
<td>0.3%</td>
<td>0.0000</td>
<td>611 FEGrid::centroid [15]</td>
</tr>
<tr>
<td></td>
<td>93.9%</td>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>[1]FEGrid::FEGrid(....)</td>
<td>0.00374</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>[2]JacobiSolver::solve</td>
<td>0.00331</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>65.3%</td>
<td>0.0022</td>
<td>202 SparseMatrix::operator* [6]</td>
</tr>
<tr>
<td></td>
<td>8.0%</td>
<td>0.0003</td>
<td>202 vector&lt;float&gt; operator- [9]</td>
</tr>
<tr>
<td></td>
<td>7.3%</td>
<td>0.0002</td>
<td>203 norm [10]</td>
</tr>
<tr>
<td></td>
<td>6.9%</td>
<td>0.0002</td>
<td>202 vector&lt;float&gt; operator+ [11]</td>
</tr>
<tr>
<td></td>
<td>5.6%</td>
<td>0.0002</td>
<td>202 vector&lt;float&gt; operator* [12]</td>
</tr>
<tr>
<td></td>
<td>0.3%</td>
<td>0.0000</td>
<td>279 SparseMatrix::operator[] const [17]</td>
</tr>
<tr>
<td></td>
<td>93.4%</td>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>[3]FEpoissonOperator::FEpoissonOperator(..)</td>
<td>0.00286</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>87.6%</td>
<td>0.0025</td>
<td>4714 buildIntegral [5]</td>
</tr>
<tr>
<td></td>
<td>87.6%</td>
<td></td>
<td>Total</td>
</tr>
</tbody>
</table>
Summary

• Machines have memory hierarchies
  • 100s of cycles to read from DRAM (main memory)
  • Caches are fast (small) memory that optimize average case
• Need to rearrange code/data to improve locality
Tips for C++

• The breadth of what is legal C++ is gigantic

• C++ Core Guideline

• I can run through some of the highlights
RAII

• Resource Acquisition Is Initialization (RAII)
  • When an object is constructed it establishes its dynamic resources
    • Typically dynamic memory (new/free)
  • Pairs with destruction is resource liberating

• Usually easiest to achieve with STL containers
  • std::vector handles acquiring and free’ing memory appropriately
  • If you have dynamic objects that have non-C++ resource acquisition, wrap it in std::shared_ptr and make that shared_ptr also have RAII

```cpp
m_multiR2C = std::shared_ptr<fftw_plan>(new fftw_plan, [] (fftw_plan* p) {
    fftw_destroy_plan(*p); delete p; });

m_multiJ = std::shared_ptr<fftw_plan>(new fftw_plan, [] (fftw_plan* p) {
    fftw_destroy_plan(*p); delete p; });
```
Rule of 0

- Five special functions for every C++ class
  - Null constructor
  - Destructor
  - Assignment
  - Move constructor
  - Move assignment
- Steal the innards of the argument object
  - Declared with && arguments

- Rule of 0
  - If you don’t define any of these, the compiler will generate all these for you if every member of the class has all five.
  - Really try and do this
  - IF you must write any of these functions yourself, then the compiler will not default any of these
    - =delete: tells the compiler to NOT define something
    - =default: tells the compiler to go ahead and try to make a default version

```cpp
class Bob
{
public:
    int m_size;
    double counter;
    std::vector<double> m_workArray;
    std::shared_ptr<Elements> m_grid;
};
```
What the heck is with move? lvalue? rvalue?

- Most hackish rule you can think about:
  - If it has a name at this scope, it is a lvalue (locator value)
  - Otherwise it is an rvalue

```c++
int Bar = Foo();
std::cout<<"the value was "<<Bar<<std::endl;

std::cout<<"the next value was "<<Foo()<<std::endl;
```

- rvalue objects are candidates for the compiler to use the move functions.

- `std::move` can turn an lvalue reference into an rvalue reference.
  - It is up to the programmer to ensure that they have done this legally
  - Rule of thumb: It is safe for the compiler to invoke the destructor immediately after `std::move` completes.