Performance Tuning and Single Processor Optimization

Expanse Webinar Series
February 18, 2021

Robert Sinkovits, PhD
Director SDSC Scientific Applications

https://github.com/sinkovit/performance-tuning
Why write efficient scalar/serial code

- **Optimizing your code will reduce your time to solution**
  - Challenging problems become doable and routine calculations can be done more quickly. You’ll be able to do more science in the same amount of time and shorten the cycle between job submission and results.

- **Computer time, particularly on high-end resources is limited**
  - If you’re running on XSEDE-allocated resources, such as *Expanse*, you need to compete with other users for access. If you’re running on the cloud, you’ll need to pay.

- **Computing uses a lot of energy**
  - Estimated that 5% of U.S. energy consumption is used to power computers.
Won’t going parallel save me?

- Most parallel applications have limited scalability
- Even if your application had perfect linear scalability, there is always a more challenging problem that you’ll want to solve
  - Higher resolution (finer grid size, shorter time step)
  - Larger systems (more atoms, molecules, particles ...)
  - More accurate physics
  - Longer simulations
  - More replicates, bigger ensembles, better statistics
- Of course availability of resources and energy usage are still important considerations (see previous slide)
Data center energy usage

In 2013, U.S. data centers consumed an estimated 91 billion kilowatt-hours of electricity, equivalent to the annual output of 34 large (500-megawatt) coal-fired power plants. **Data center electricity consumption is projected to increase to roughly 140 billion kilowatt-hours annually by 2020, the equivalent annual output of 50 power plants,** costing American businesses $13 billion annually in electricity bills and emitting nearly 100 million metric tons of carbon pollution per year.

Guidelines for software optimization

• Don’t break your code – make sure you still get correct results
• Profile your code so that you know where to focus your effort
• Don’t obfuscate your code unless you have a really compelling reason (e.g. changes to kernel in heavily use application)
• Document your changes, especially if the new code looks significantly different from the original
• Use optimized libraries when possible
• Understand the capabilities and limitations of your compiler
How much effort should you put into optimizing your code?

The amount of time and effort you spend on optimizing your code depends on a number of factors

• Is the code heavily/widely used?
• Does the code consume a significant amount of computer time?
• Is time to solution important?
• Will optimizing your code help you solve new sets of problems?
• Have you reached the point where most of the computing occurs in routines outside your control?
• Would your time be better spent doing your research?
Profile your code before you dive in!

Modern software can contain many thousand lines of code. Profile before you get started so that you know where to focus your effort. Very often, most of the time is spent in a small number of routines.

The venerable gprof tool (1982) is a good place to start

- Universal support by all major C/C++ and Fortran compilers
- Very easy to use: compile with –pg flag; run code; gprof a.out gmon.out
- Introduces virtually no overhead

Note that gprof is not a tracing tool and won’t identify bottlenecks in parallel codes. Still, it’s a great choice for understanding behavior of underlying serial code.
The gprof flat profile is a simple listing of functions/subroutines ordered by their relative usage. Often a small number of routines will account for a large majority of the run time. Useful for identifying hot spots in your code.

### Flat profile:

<table>
<thead>
<tr>
<th>%</th>
<th>cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
<td>ms/call</td>
<td>ms/call</td>
</tr>
<tr>
<td>68.60</td>
<td>574.72</td>
<td>574.72</td>
<td>399587</td>
<td>1.44</td>
<td>1.44</td>
</tr>
<tr>
<td>13.48</td>
<td>687.62</td>
<td>112.90</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.60</td>
<td>784.81</td>
<td>97.19</td>
<td>182889</td>
<td>0.53</td>
<td>0.53</td>
</tr>
<tr>
<td>2.15</td>
<td>802.85</td>
<td>18.04</td>
<td>182889</td>
<td>0.10</td>
<td>0.63</td>
</tr>
<tr>
<td>1.52</td>
<td>815.56</td>
<td>12.71</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.28</td>
<td>826.29</td>
<td>10.73</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.96</td>
<td>834.30</td>
<td>8.02</td>
<td>22183</td>
<td>0.36</td>
<td>0.36</td>
</tr>
<tr>
<td>0.12</td>
<td>835.27</td>
<td>0.97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.08</td>
<td>835.94</td>
<td>0.66</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.06</td>
<td>836.45</td>
<td>0.51</td>
<td>22183</td>
<td>0.02</td>
<td>5.58</td>
</tr>
<tr>
<td>0.05</td>
<td>836.88</td>
<td>0.44</td>
<td>1</td>
<td>436.25</td>
<td>436.25</td>
</tr>
</tbody>
</table>
gprof call graph
The gprof call graph provides additional levels of detail such as the exclusive time spent in a function, the time spent in all children (functions that are called) and statistics on calls from the parent(s)

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>96.9</td>
<td>112.9</td>
<td>699.04</td>
<td></td>
<td>main [1]</td>
</tr>
<tr>
<td></td>
<td>574.72</td>
<td>0.00</td>
<td>399587/399587</td>
<td></td>
<td>get_number_packed_data [2]</td>
</tr>
<tr>
<td></td>
<td>0.51</td>
<td>123.25</td>
<td>22183/22183</td>
<td></td>
<td>Is_Hump [3]</td>
</tr>
<tr>
<td></td>
<td>0.44</td>
<td>0.00</td>
<td>1/1</td>
<td></td>
<td>quickSort [11]</td>
</tr>
<tr>
<td></td>
<td>0.04</td>
<td>0.00</td>
<td>1/1</td>
<td></td>
<td>radixsort_flock [18]</td>
</tr>
<tr>
<td></td>
<td>0.02</td>
<td>0.00</td>
<td>2/2</td>
<td></td>
<td>ID2Center_all [19]</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>------</td>
<td>----------</td>
<td>--------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td></td>
<td>574.72</td>
<td>0.00</td>
<td>399587/399587</td>
<td></td>
<td>main [1]</td>
</tr>
<tr>
<td>[2]</td>
<td>68.6</td>
<td>574.72</td>
<td>0.00</td>
<td>399587</td>
<td>get_number_packed_data [2]</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>------</td>
<td>----------</td>
<td>--------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td></td>
<td>0.51</td>
<td>123.25</td>
<td>22183/22183</td>
<td></td>
<td>main [1]</td>
</tr>
<tr>
<td>[3]</td>
<td>14.8</td>
<td>574.72</td>
<td>0.00</td>
<td>399587</td>
<td>get_nearest_events [4]</td>
</tr>
<tr>
<td></td>
<td>18.04</td>
<td>97.19</td>
<td>182889/182889</td>
<td></td>
<td>pack_arrays [8]</td>
</tr>
<tr>
<td></td>
<td>8.02</td>
<td>0.00</td>
<td>22183/22183</td>
<td></td>
<td>pack_points [24]</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>22183/22183</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Value of reprofiling

Be sure to reprofile your code after you’ve done a round of optimization since new hotspots may emerge. In this case, \texttt{get\_number\_packed\_data} has been improved so much that \texttt{main} is now the most time-consuming routine.

<table>
<thead>
<tr>
<th>time</th>
<th>seconds</th>
<th>seconds</th>
<th>calls</th>
<th>ms/call</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>41.58</td>
<td>36.95</td>
<td>36.95</td>
<td></td>
<td></td>
<td></td>
<td>main</td>
</tr>
<tr>
<td>26.41</td>
<td>60.42</td>
<td>23.47</td>
<td>22183</td>
<td>1.06</td>
<td>1.06</td>
<td>\texttt{get_number_packed_data}</td>
</tr>
<tr>
<td>11.58</td>
<td>70.71</td>
<td>10.29</td>
<td></td>
<td></td>
<td></td>
<td>\texttt{_c_mcopy8}</td>
</tr>
<tr>
<td>10.98</td>
<td>80.47</td>
<td>9.76</td>
<td>182889</td>
<td>0.05</td>
<td>0.05</td>
<td>\texttt{get_nearest_events}</td>
</tr>
<tr>
<td>8.43</td>
<td>87.96</td>
<td>7.49</td>
<td>22183</td>
<td>0.34</td>
<td>0.34</td>
<td>\texttt{pack_arrays}</td>
</tr>
<tr>
<td>0.57</td>
<td>88.47</td>
<td>0.51</td>
<td>22183</td>
<td>0.02</td>
<td>0.80</td>
<td>\texttt{Is_Hump}</td>
</tr>
<tr>
<td>0.20</td>
<td>88.65</td>
<td>0.18</td>
<td>1</td>
<td>180.00</td>
<td>180.00</td>
<td>\texttt{quickSort}</td>
</tr>
<tr>
<td>0.08</td>
<td>88.72</td>
<td>0.07</td>
<td></td>
<td></td>
<td></td>
<td>\texttt{_init}</td>
</tr>
<tr>
<td>0.05</td>
<td>88.76</td>
<td>0.04</td>
<td>1</td>
<td>40.00</td>
<td>40.00</td>
<td>\texttt{radixsort_flock}</td>
</tr>
<tr>
<td>0.02</td>
<td>88.78</td>
<td>0.02</td>
<td>1</td>
<td>20.00</td>
<td>20.00</td>
<td>\texttt{compute_position}</td>
</tr>
<tr>
<td>0.02</td>
<td>88.80</td>
<td>0.02</td>
<td>1</td>
<td>20.00</td>
<td>20.00</td>
<td>\texttt{readsource}</td>
</tr>
</tbody>
</table>

Previously used 13% of wall time
Previously used 68% of wall time
Value of reprofiling

Be sure to reprofile your code after you’ve done a round of optimization since new hotspots may emerge. In this case, `get_number_packed_data` has been improved so much that `main` is now the most time-consuming routine.

<table>
<thead>
<tr>
<th>time</th>
<th>seconds</th>
<th>seconds</th>
<th>calls</th>
<th>ms/call</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>41.58</td>
<td>36.95</td>
<td>36.95</td>
<td></td>
<td></td>
<td></td>
<td>main</td>
</tr>
<tr>
<td>26.41</td>
<td>60.42</td>
<td>23.47</td>
<td>22183</td>
<td>1.06</td>
<td>1.06</td>
<td><code>get_number_packed_data</code></td>
</tr>
<tr>
<td>11.58</td>
<td>70.71</td>
<td>10.29</td>
<td></td>
<td></td>
<td></td>
<td><code>__c_mcopy8</code></td>
</tr>
<tr>
<td>10.98</td>
<td>80.47</td>
<td>9.76</td>
<td>182889</td>
<td>0.05</td>
<td>0.05</td>
<td><code>get_nearest_events</code></td>
</tr>
<tr>
<td>8.43</td>
<td>87.96</td>
<td>7.49</td>
<td>22183</td>
<td>0.34</td>
<td>0.34</td>
<td><code>pack_arrays</code></td>
</tr>
<tr>
<td>0.57</td>
<td>88.47</td>
<td>0.51</td>
<td>22183</td>
<td>0.02</td>
<td>0.80</td>
<td><code>Is_Hump</code></td>
</tr>
<tr>
<td>0.20</td>
<td>88.65</td>
<td>0.18</td>
<td>1</td>
<td>180.00</td>
<td>180.00</td>
<td><code>quickSort</code></td>
</tr>
<tr>
<td>0.08</td>
<td>88.72</td>
<td>0.07</td>
<td></td>
<td></td>
<td></td>
<td><code>_init</code></td>
</tr>
<tr>
<td>0.05</td>
<td>88.76</td>
<td>0.04</td>
<td>1</td>
<td>40.00</td>
<td>40.00</td>
<td><code>radixsort_flock</code></td>
</tr>
<tr>
<td>0.02</td>
<td>88.78</td>
<td>0.02</td>
<td>1</td>
<td>20.00</td>
<td>20.00</td>
<td><code>compute_position</code></td>
</tr>
<tr>
<td>0.02</td>
<td>88.80</td>
<td>0.02</td>
<td>1</td>
<td>20.00</td>
<td>20.00</td>
<td><code>readsource</code></td>
</tr>
</tbody>
</table>
It’s been 40 years since gprof, is there anything new?

The AMD µProf tool can provide much more detail than gprof, including low-level information on cache misses, branch mispredictions, etc. µProf will be a future training topic.
Compute bound vs. memory bound codes

**Memory-bound codes**: rate at which data can be delivered to the CPU is the limiting factor. Goal will be to apply cache-level optimizations so that the CPU is not starved for data.

**Compute-bound codes**: performance of the processor is limiting factor. Data can be delivered fast enough, but the processor can’t keep up. Our goal will be to reduce the amount of computation done on a given piece of data.

In real applications, we’ll often deal with a combination of compute- and memory-bound kernels.
Computer memory hierarchy

- **Registers**: Fast, Small, $$$$
- **L1 cache**: O(ns), O(10 KB)
- **L2 cache**: O(10 ns), O(100 KB)
- **L3 cache**: O(10 ns), O(10 MB)
- **DRAM**: O(100 ns), O(10-100 GB)
- **Disk**: O(100 μs SSD), O(ms HDD), O(TB - PB)

**Comparison**
- **Fast** vs. **Slow**
- **Small** vs. **Large**
- **Expensive** vs. **Cheap**
Cache essentials and memory bound codes

**Temporal Locality:** Data that was recently accessed is likely to be used again in the near future. To take advantage of temporal locality, once data is loaded into cache, it will generally remain there until it has to be purged to make room for new data. Cache is typically managed using a variation of the Least Recently Used (LRU) algorithm.

**Spatial locality:** If a piece of data is accessed, it’s likely that neighboring data elements in memory will be needed. To take advantage of spatial locality, cache is organized into lines (typically 64 bytes) and an entire line is loaded at once.

Our goal in cache level optimization is very simple – exploit the principles of temporal and spatial locality to minimize data access times.
One-dimensional arrays

One-dimensional arrays are stored as blocks of contiguous data in memory.

```c
int *x, n=100;
x = (int *) malloc(n * sizeof(int))
```

Cache optimization for 1D arrays is straightforward and you’ll probably write optimal code without even trying. When possible, just access the elements in order.

```c
for (int i=0; i<n; i++) {
    x[i] += 100;
}
```
One-dimensional arrays

What is our block of code doing with regards to cache?

for (int i=0; i<n; i++) {
    x[i] += 100;
}

Assuming a 64-byte cache line and 4-byte integers:

1. Load elements x[0] through x[15] into cache
2. Increment x[0] through x[15]
3. Load elements x[16] through x[31] into cache
4. Increment elements x[16] through x[31]
5. ...

In reality, the processor will recognize the pattern of data access and prefetch the next cache line before it is needed
Do I have control over cache?

There are no programming construct that I’m aware of that give you direct control over cache (e.g. load a particular location in memory into cache).

Modern processors directly implement advanced cache replacement strategies, branch prediction and prefetch mechanisms. The best you can do is to follow standard practices to exploit temporal and spatial locality and, in some instances, choose optimal parameters based on the cache sizes.
Multidimensional arrays

From the computer’s point of view, there is no such thing as a two-dimensional array. This is just syntactic sugar provided as a convenience to the programmer. Under the hood, array is stored as linear block of data.

**Column-major order**: First or leftmost index varies the fastest. Used in Fortran, R, and MATLAB

![Column-major order example]

**Row-major order**: Last or rightmost index varies the fastest. Used in Python, Mathematica and C/C++

![Row-major order example]
Multidimensional arrays – array addition example

Properly written Fortran code (leftmost index varies fastest)

    do j=1,n ! Note loop nesting
        do i=1,n
            z(i,j) = x(i,j) + y(i,j)
        enddo
    enddo

Properly written C code (rightmost index varies fastest)

    for (i=0; i<n; i++) { // Note loop nesting
        for (j=0; j<n; j++) {
            z[i][j] = x[i][j] + y[i][j]
        }
    }
Compiling codes

In the following examples, we’ll compile the C or Fortran codes using the Intel compilers (icc/ifort) and AMD AOCC compilers (clang/flang)

We’re mainly concerned with the compiler flags that control the overall optimization level and the target architecture

<table>
<thead>
<tr>
<th>optimization</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>varies</td>
</tr>
<tr>
<td>-O0</td>
<td>no optimization</td>
</tr>
<tr>
<td>-O1</td>
<td>limited optimization</td>
</tr>
<tr>
<td>-O2</td>
<td>moderate</td>
</tr>
<tr>
<td>-O3</td>
<td>aggressive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>compiler</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD clang/flag</td>
<td>-march=znver2</td>
</tr>
<tr>
<td>Intel icc/ifort</td>
<td>-march=core-avx2</td>
</tr>
</tbody>
</table>
Is data access order really that important?

Run times for adding two matrices of size 30,000 x 30,000 using a Fortran code with loops nested in the proper and improper orders. This is a memory-bound application since we’re only doing one addition for every two 8-byte floats that we load.

<table>
<thead>
<tr>
<th>optimization</th>
<th>AOCC (flang) proper</th>
<th>AOCC (flang) improper</th>
<th>Intel (ifort) proper</th>
<th>Intel (ifort) improper</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>3.47</td>
<td>9.19</td>
<td>default</td>
<td>1.19</td>
</tr>
<tr>
<td>-O0</td>
<td>3.47</td>
<td>9.19</td>
<td>-O0</td>
<td>6.67</td>
</tr>
<tr>
<td>-O1</td>
<td>1.68</td>
<td>5.11</td>
<td>-O1</td>
<td>1.25</td>
</tr>
<tr>
<td>-O2</td>
<td>1.14</td>
<td>4.94</td>
<td>-O2</td>
<td>1.19</td>
</tr>
<tr>
<td>-O3</td>
<td>1.14</td>
<td>4.94</td>
<td>-O3</td>
<td>1.19</td>
</tr>
</tbody>
</table>
AOCC compiler – what happened?

AOCC (flang)

<table>
<thead>
<tr>
<th>optimization</th>
<th>proper</th>
<th>improper</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>3.47</td>
<td>9.19</td>
</tr>
<tr>
<td>-O0</td>
<td>3.47</td>
<td>9.19</td>
</tr>
<tr>
<td>-O1</td>
<td>1.68</td>
<td>5.11</td>
</tr>
<tr>
<td>-O2</td>
<td>1.14</td>
<td>4.94</td>
</tr>
<tr>
<td>-O3</td>
<td>1.14</td>
<td>4.94</td>
</tr>
</tbody>
</table>

Run times improve as we enable higher levels of optimization (up to -O2)

Default appears to be the same as disabling all optimizations (-O0)

Improper loop nesting is always slower than proper loop nesting at all optimization levels
Intel compiler – what happened?

Run times improve as we enable higher levels of optimization (up to -O2)

Default appears to be the same as -O2

Intel compiler is “smart” enough to recognize that the loops are improperly nested and reorders to get better performance

<table>
<thead>
<tr>
<th>optimization</th>
<th>proper</th>
<th>improper</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>1.19</td>
<td>1.19</td>
</tr>
<tr>
<td>-O0</td>
<td>6.67</td>
<td>14.88</td>
</tr>
<tr>
<td>-O1</td>
<td>1.25</td>
<td>4.91</td>
</tr>
<tr>
<td>-O2</td>
<td>1.19</td>
<td>1.19</td>
</tr>
<tr>
<td>-O3</td>
<td>1.19</td>
<td>1.19</td>
</tr>
</tbody>
</table>
Lessons

- Making optimal use of cache is particularly important in memory bound codes.
- Sometimes the compiler will help you (Intel) and sometimes it won’t (AOCC)

<table>
<thead>
<tr>
<th>optimization</th>
<th>proper</th>
<th>improper</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>3.47</td>
<td>9.19</td>
</tr>
<tr>
<td>-00</td>
<td>3.47</td>
<td>9.19</td>
</tr>
<tr>
<td>-01</td>
<td>1.68</td>
<td>5.11</td>
</tr>
<tr>
<td>-02</td>
<td>1.14</td>
<td>4.94</td>
</tr>
<tr>
<td>-03</td>
<td>1.14</td>
<td>4.94</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>optimization</th>
<th>proper</th>
<th>improper</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>1.19</td>
<td>1.19</td>
</tr>
<tr>
<td>-00</td>
<td>6.67</td>
<td>14.88</td>
</tr>
<tr>
<td>-01</td>
<td>1.25</td>
<td>4.91</td>
</tr>
<tr>
<td>-02</td>
<td>1.19</td>
<td>1.19</td>
</tr>
<tr>
<td>-03</td>
<td>1.19</td>
<td>1.19</td>
</tr>
</tbody>
</table>
More loop level optimizations

- Loop fusion
- Loop fission
- Loop-invariant code motion
- Loop peeling
- Loop unrolling
- Loop blocking
- Breaking out of loops early
- Short loop optimizations
Loop fusion

One of the most basic loop-level optimizations is loop fusion. Two or more loops with the same range of iterations are combined into a single loop.

Fusing simple loops give the compiler more opportunities to rearrange instructions and get better performance. Even more beneficial if the fused loops use the same data streams since it allows for better data reuse.
Results of loop fusion experiment

Benchmarks on case illustrate on previous slide for fused and unfused loops with 900 million array elements. For a given compiler and optimization level, the fused loop is always faster.

<table>
<thead>
<tr>
<th>optimization</th>
<th>AOCC (flang)</th>
<th>Intel (ifort)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fused</td>
<td>unfused</td>
</tr>
<tr>
<td>default</td>
<td>6.36</td>
<td>7.03</td>
</tr>
<tr>
<td>-00</td>
<td>6.34</td>
<td>6.95</td>
</tr>
<tr>
<td>-01</td>
<td>3.56</td>
<td>4.54</td>
</tr>
<tr>
<td>-02</td>
<td>2.43</td>
<td>3.24</td>
</tr>
<tr>
<td>-03</td>
<td>2.43</td>
<td>3.24</td>
</tr>
</tbody>
</table>
Loop fission

Loop fission is the opposite of loop fusion. Sometimes if the loop body is too complex or contains too many data streams, splitting can improve performance.

It’s often difficult to decide if loops should be fused or split. If in doubt, try both versions and see which is faster.
Loop-invariant code motion

Pull an invariant calculation out of a loop and use the pre-calculated result in its place. Although compilers can often do this for you if the loop is simple, we recommend doing this yourself. No real downsides and potentially big savings.

```
for (int i=0; i<n; i++) { 
    z[i] = x[i] + sqrt(c);
}
```

```
sqrtc = sqrt(c);
for (int i=0; i<n; i++) { 
    z[i] = x[i] + sqrtc;
}
```
Loop-invariant code motion – a more complex example

When working with nested loops, the invariants will sometime be less obvious and may even be a vector of results.

```c
for (i=0; i<nx; i++) {
    for (j=0; j<ny; j++) {
        for (k=0; k<nz; k++) {
            x2y2 = x[i]*x[i] + y[j]*y[j];
            z2   = z[k] * z[k];
            res[i][j][k] = exp(-a*z2) * sqrt(b*x2y2);
        }
    }
}
```

- `x2y2` does not depend on index `k`
- `sqrt(b*x2y2)` does not depend on index `k`
Loop-invariant code motion – a more complex example

1. Move \( x^2 \) to outermost loop nesting; evaluated \( nx \) times instead of \( nx*ny*nz \)
2. Move calculation of \( \sqrt{x^2+y^2} \) out one level; evaluated \( nx*ny \) times rather than \( nx*ny*nz \)

```c
for (i=0; i<nx; i++) {
    x2 = x[i]*x[i];
    for (j=0; j<ny; j++) {
        x2y2 = x2 + y[j]*y[j];
        sqrtx2y2 = sqrt(b*x2y2);
        for (k=0; k<nz; k++) {
            z2   = z[k] * z[k];
            res[i][j][k] = exp(-a*z2) * sqrtx2y2;
        }
    }
}
```
Loop-invariant code motion – a more complex example

Pre-calculate vector of \( \exp(-a*z^2) \) results and reuse for every set of \((i,j)\). Reduces number of exponential evaluations to \( nz \) from \( nx*ny*nz \)

```c
for (k=0; k<nz; k++) {
    zterm[k] = exp(-a*z[k]*z[k]);
}

for (i=0; i<nx; i++) {
    x2 = x[i]*x[i];
    for (j=0; j<ny; j++) {
        x2y2 = x2 + y[j]*y[j];
        sqrtx2y2 = sqrt(b*x2y2);
        for (k=0; k<nz; k++) {
            res[i][j][k] = zterm[k] * sqrtx2y2;
        }
    }
}
```
Loop peeling

In a loop peeling optimization, one or more iterations are pulled out of the loop. Avoids unnecessary calculations associated with special iterations; also allows fusion of loops with slightly different iteration ranges.

```java
for (int i=0; i<n; i++) {
    if (i == 0) {
        z[i] = x[i] / y[i];
    } else {
        z[i] = x[i] + y[i];
    }
}
for (int i=1; i<n; i++) {
    w[i] = x[i] * y[i]
}
```

This example illustrates how peeling off the first iteration of the first loop (i=0) both avoids special case (product instead of sum) and allows fusion with the following loop.

```java
for (int i=0; i<n; i++) {
    z[0] = x[0] / y[0];
    for (int i=1; i<n; i++) {
        z[i] = x[i] + y[i]
        w[i] = x[i] * y[i]
    }
```
Loop unrolling

Loop body is replicated and the stride is modified accordingly. This optimization can help the processor make better use of arithmetic functional units.

```java
for (int i=0; i<1024; i++) {
    z[i] = x[i] + y[i]
}
```

```plaintext
for (int i=0; i<1024; i+=4) {
    z[i]   = x[i]   + y[i]
    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]
}
```

This example is particularly simple since the loop count is divisible by the unrolling depth. In general, you’ll need to write cleanup code to handle the leftover iterations (remainder of n/depth).

**WARNING**: you will rarely beat the compiler and manual loop unrolling will make your code ugly and difficult to maintain. Best choice for unrolling depth may be processor architecture dependent.
Loop unrolling

Although you’ll rarely beat the compiler, sometimes you’ll encounter a loop that is too complex for it to accurately analyze. Below is an example where manual loop unrolling by 4x did better than the compiler (original loop shown)

```
do i=0,4319,2 ! Unrolled loop → i=0,4319,8
   j0=mg63_mijjj(0,i)
   j1=mg63_mijjj(1,i)
   j2=mg63_mijjj(2,i)
   i0=mg63_mijjj(3,i)
   i1=mg63_mijjj(4,i)
   i2=mg63_mijjj(5,i)
   i3=mg63_mijjj(6,i)
   pvi3jjj(1 ) = pvi3jjj(1 ) + d(i0,i1)*d(i0,i2)*d(i0,i3)*d(i0,j0)*d(i0,j1)
   pvi3jjj(2 ) = pvi3jjj(2 ) + d(i0,i1)*d(i1,i2)*d(i0,i3)*d(i0,j0)*d(i0,j1)
   pvi3jjj(3 ) = pvi3jjj(3 ) + d(i0,i1)*d(i1,i2)*d(i1,i3)*d(i0,j0)*d(i0,j1)
   [ Several hundred lines of code not shown ]
   pvi3jjj(22) = pvi3jjj(22) + d(i0,i2)*d(i0,i3)*d(i0,j0)*d(i1,j0)*d(j0,j1)
   pvi3jjj(23) = pvi3jjj(23) + d(i1,i2)*d(i0,i3)*d(i0,j0)*d(i1,j0)*d(j0,j1)
   pvi3jjj(24) = pvi3jjj(24) + d(i0,i2)*d(i2,i3)*d(i0,j0)*d(i1,j0)*d(j0,j1)
enddo
```
Breaking out of loop early

Look for opportunities to break out of a loop early. This will generally require that you understand the semantics of your code.

```c
for (int i=0; i<n; i++) {
    if (y[i] < const) {
        // Do stuff
    }
}
```

```c
for (int i=0; i<n; i++) {
    if (y[i] >= const) {
        break;
    } else {
        // Do stuff
    }
}
```

In this simplified example (taken from real-life finance application), I used my knowledge that the elements of array y are monotonically increasing ($y[0] \leq y[1] \leq y[2] \leq y[3] \ldots$). The compiler only understands the syntax of your code and cannot safely do this optimization for you.
Optimize for the common case

The Model for Prediction Across Scales (MPAS) is a collaborative project between NCAR and LANL for developing atmosphere, ocean and other earth-system simulation components for use in global climate, regional climate and weather studies.

The unstructured grid consists overwhelmingly of hexagons, plus a very small number of pentagons and heptagons. We were able to speed up one of the key routines by optimizing for the common case.

Above: Centroidal Voronoi tessellations provide conformal meshes with smooth transitions between regions of differing resolution.
Optimize for the common case

The Model for Prediction Across Scales (MPAS) is a collaborative project between NCAR and LANL for developing atmosphere, ocean and other earth-system simulation components for use in global climate, regional climate and weather studies.

```fortran
select case(nEdgesOnCell(iCell))
    case(6)
        do k=1, nVertLevels
            s_max(k,iCell) = max(s_max(k,iCell),
            scalar_old(k, cellsOnCell(1,iCell)),
            scalar_old(k, cellsOnCell(2,iCell)),
            scalar_old(k, cellsOnCell(3,iCell)),
            scalar_old(k, cellsOnCell(4,iCell)),
            scalar_old(k, cellsOnCell(5,iCell)),
            scalar_old(k, cellsOnCell(6,iCell)))
        end do
    end select
    case default
        do i=1, nEdgesOnCell(iCell)
            do k=1, nVertLevels
                s_max(k,iCell) = max(s_max(k,iCell),scalar_old(k, cellsOnCell(i,iCell)))
            end do
        end do
end select
```

Split off and optimized the case where the cell is hexagonal

In original code, looped over the number of edges regardless of the cell shape
Force reductions – power functions

A force reduction optimization involves the replacement of an expensive operation with an equivalent, less expensive one.

Exponentiation operations, especially floating point base raised to a floating point power, are particularly expensive. Look for opportunities to replace with multiplications, particularly if the exponent is known at compile time

\[
\begin{align*}
\text{pow}(x, 8) & \rightarrow x2 = x*x; x4 = x2*x2; x8 = x4*x4 \\
\text{pow}(x, 1.5) & \rightarrow y = x \times \text{sqrt}(x)
\end{align*}
\]

Many languages overload the power function. If you really intend to raise to an integer power, be sure to use an integer argument.
Force reductions – trig functions

If code spends a lot of time evaluating trig functions, may have the potential for a big payoff by applying your high school math. Just be sure that identities apply to all quadrants if applicable.

\[
\begin{align*}
\sin(x)\cos(x) & \rightarrow 0.5 \times \sin(2x) \\
\sin(x)\cos(y) + \cos(x)\sin(y) & \rightarrow \sin(x+y)
\end{align*}
\]

If \(a\) and \(b\) are fixed and sum needs to be calculated repeatedly for many values of \(x\), can pre-calculate the constants \(c\) and \(\phi\).

\[
\begin{align*}
a\sin(x) + b\cos(x) & \rightarrow c = \sqrt{a^2 + b^2} \\
\phi & = \text{atan2}(b,a) \\
c\sin(x+\phi)
\end{align*}
\]
Force reductions – hidden opportunities

There are often hidden opportunities for force reductions. Look at logical tests that can be written in a more efficient way. Think about what results are really needed.

count = 0;
for (i=0; i<n; i++) {
    if (log(x[i]) < c) {
        count++;
    }
}

count = 0;
expc = exp(c)
for (i=0; i<n; i++) {
    if (x[i] < expc) {
        count++;
    }
}

In this example, we didn’t really need to know the logarithm of x[i] and we could recast using a simple comparison to a pre-computed value.
Force reduction programming example

Results of toy program that randomly generates N (100,000) particles in a unit square and then tests by brute force how many particle pairs are separated by less than a specified distance (0.01)

- With force reduction tests \((x^2+y^2) \leq d^2\)
- Without force reduction tests \(\sqrt{x^2+y^2} \leq d\)

<table>
<thead>
<tr>
<th>conditions</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>clang w/ force reduction</td>
<td>1.18</td>
</tr>
<tr>
<td>clang w/o force reduction</td>
<td>3.20</td>
</tr>
<tr>
<td>icc w/ force reduction</td>
<td>1.22</td>
</tr>
<tr>
<td>icc w/o force reduction</td>
<td>3.43</td>
</tr>
</tbody>
</table>
Inter-procedural optimizations

• Inter-procedural optimizations are based on a high-level view of the entire program and span multiple functions

• Compilers are great at optimizing loops (inversion, unrolling, fusion, splitting, peeling, etc.) and statements, but can rarely recognize opportunities for inter-procedural optimizations.

• These generally require an intimate understanding of your code.

• Very often, these optimizations depend on recognizing operations that are repeated on the same set of data from one invocation of a function to the next.
Inter-procedural optimizations – flow cytometry example

Flow cytometry is a laboratory technique used to characterize cells based on the molecules that they express on their surfaces. Sorting these cells into different populations is a computationally challenging problem.


Inter-procedural optimizations – flow cytometry example

Working with application from J. Craig Venter Institute (JCVI), recognized that most time-consuming function was called five times with slightly different arguments

\[
\begin{align*}
E_i &= \text{get_avg_dist}(\text{rpc[temp_i]}, \text{temp_i}, \text{temp_j}, \text{population_ID}, \text{num_real_pop}, \text{file_len}, \text{num_dm}, \text{norm_data}, d1, d2, d3, \text{size_c}); \\
E_j &= \text{get_avg_dist}(\text{rpc[temp_j]}, \text{temp_i}, \text{temp_j}, \text{population_ID}, \text{num_real_pop}, \text{file_len}, \text{num_dm}, \text{norm_data}, d1, d2, d3, \text{size_c}); \\
E_1 &= \text{get_avg_dist}(\text{center_1}, \text{temp_i}, \text{temp_j}, \text{population_ID}, \text{num_real_pop}, \text{file_len}, \text{num_dm}, \text{norm_data}, d1, d2, d3, \text{size_c}); \\
E_2 &= \text{get_avg_dist}(\text{center_2}, \text{temp_i}, \text{temp_j}, \text{population_ID}, \text{num_real_pop}, \text{file_len}, \text{num_dm}, \text{norm_data}, d1, d2, d3, \text{size_c}); \\
E_3 &= \text{get_avg_dist}(\text{center_3}, \text{temp_i}, \text{temp_j}, \text{population_ID}, \text{num_real_pop}, \text{file_len}, \text{num_dm}, \text{norm_data}, d1, d2, d3, \text{size_c});
\end{align*}
\]
Inter-procedural optimizations – flow cytometry example

Within the get_avg_dist function, the key loops involve a comparison between elements of population_ID and the scalars (temp_i, temp_j) to decide which elements of norm_data are used for the calculations. Recall that center is the only argument to change between calls and the same elements of norm_data are used all five times.

```c
get_avg_dist(center, temp_i, temp_j, population_ID,
            num_real_pop, file_len, num_dm, norm_data, d1, d2, d3, size_c);

for (i=0; i<file_len; i++) {
    if (population_ID[i]==temp_i || population_ID[i]==temp_j) {
        dist1 = center[d1] - norm_data[i][d1];
        dist2 = center[d2] - norm_data[i][d2];
        dist3 = center[d3] - norm_data[i][d3];
        d = dist1*dist1 + dist2*dist2 + dist3*dist3;
        if (d < radius) num_neighbors++
    }
}
Inter-procedural optimizations – flow cytometry example

To avoid having to do the same tests five times in a row, do a “gather” operation to collect elements of packed data into an array and pass as argument to a modified `get_avg_dist`. Led to ~ 3x speedup of program.

```c
npacked = 0;
for (i=0;i<file_len;i++) {
    if (population_ID[i]==temp_i || population_ID[i]==temp_j){
        packed1[npacked] = norm_data[i][d1];
        packed2[npacked] = norm_data[i][d2];
        packed3[npacked] = norm_data[i][d3];
        npacked++;
    }
}
for (i=0; i<npacked; i++) {
    dist1 = center[d1] - packed1[i];
    dist2 = center[d2] - packed2[i];
    dist3 = center[d3] - packed3[i];
    d = dist1*dist1 + dist2*dist2 + dist3*dist3;
    if (d < radius) num_neighbors++
}
```
Summary

• Optimizing your code reduces time to solution, saves energy and make resources go further
• Before you get started, make sure it’s worth your effort
• Optimization can change results, ask yourself ‘how critical is reproducibility?’
• Profile, optimize, repeat …
• Take advantage of optimized libraries and the work of others
  • Good programmers write good code
  • Great programmers steal great code
• Know the capabilities and limitations of your compiler, but don’t rely on the compiler to fix all your bad programming practices
• Optimizing for cache is critical – exploit spatial & temporal locality
• The biggest payoffs often come from a deep understanding of the semantics and structure of your code