CPU profiling on Expanse

HPC User Training
April 22, 2022

Robert Sinkovits, PhD
Director SDSC Scientific Applications

https://github.com/sinkovit/profiling-tutorial

Why should you profile your code?

- **Determine what portions of your code are using the most time**
  - Modern HPC and data intensive software often comprises many thousands of lines of code. Before you start trying to improve the performance, you need to know where to spend your effort.
- **Figure out why those portions of your code take so much time**
  - Understanding why a section of code is so time consuming can provide valuable insights into how it can be improved.

Profiling tools: gprof

- Long history going all the way back to 1982
- Non-proprietary; not tied to any specific vendor or architecture
- Available everywhere
- Universal support by all major C/C++ and Fortran compilers
- Very easy to use:
  - compile with -pg flag
  - run code to generate gmon.out file
  - gprof a.out gmon.out
- Introduces virtually no overhead

gprof flat profile

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:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 68.60    574.72   574.72   399587     1.44     1.44  get_number_packed_data
 13.48    687.62   112.90                             main
 11.60    784.81    97.19   182889     0.53     0.53  quickSort_double
  2.15    802.85    18.04   182889     0.10     0.63  get_nearest_events
  1.52    815.56    12.71                             __c_mcopy8
  1.28    826.29    10.73                             _mcount2
  0.96    834.30     8.02    22183     0.36     0.36  pack_arrays
  0.12    835.27     0.97                             __rouexit
  0.08    835.94     0.66                             __rouinit
  0.06    836.45     0.51    22183     0.02     5.58  Is_Hump
  0.05    836.88     0.44        1   436.25   436.25  quickSort
```

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)

```
index % time    self  children    called     name
[1]     96.9  112.90  699.04                 main [1]
              574.72    0.00  399587/399587      get_number_packed_data [2]
                0.51  123.25   22183/22183       Is_Hump [3]
                0.44    0.00       1/1           quickSort [11]
                0.04    0.00       1/1           radixsort_flock [18]
                0.02    0.00       2/2           ID2Center_all [19]
-----------------------------------------------
              574.72    0.00  399587/399587      main [1]
[2]     68.6  574.72    0.00  399587         get_number_packed_data [2]
-----------------------------------------------
                0.51  123.25   22183/22183       main [1]
[3]     14.8    0.51  123.25   22183         Is_Hump [3]
               18.04   97.19  182889/182889      get_nearest_events [4]
                8.02    0.00   22183/22183       pack_arrays [8]
                0.00    0.00   22183/22183       pack_points [24]
```

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.

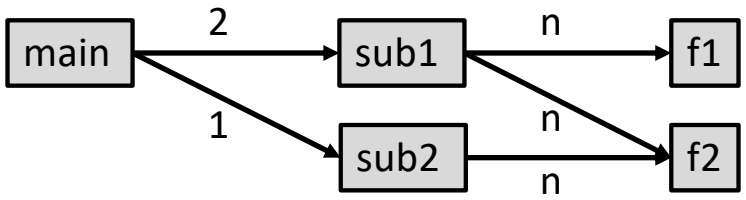
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 41.58     36.95    36.95                             main
 26.41     60.42    23.47    22183     1.06     1.06  get_number_packed_data
 11.58     70.71    10.29                             __c_mcopy8
 10.98     80.47     9.76   182889     0.05     0.05  get_nearest_events
  8.43     87.96     7.49    22183     0.34     0.34  pack_arrays
  0.57     88.47     0.51    22183     0.02     0.80  Is_Hump
  0.20     88.65     0.18        1   180.00   180.00  quickSort
  0.08     88.72     0.07                             _init
  0.05     88.76     0.04        1    40.00    40.00  radixsort_flock
  0.02     88.78     0.02        1    20.00    20.00  compute_position
  0.02     88.80     0.02        1    20.00    20.00  readsource
```

Previously used 13% of wall time

Previously used 68% of wall time

A simple gprof example



- Main program call sub1 twice and sub2 once.
- sub1 calls f1 and f2 n times
- sub2 calls f1 n times

Build: ifort -pg -march=core-avx2 -O3 -o intro intro.f

Run: time ./intro 1000000000

Profile: gprof intro gmon.out > profile_intro

A simple gprof example

Results don't look like what we expected. The run time reported by gprof matches what we measured (18 s), but 100% of the run time is attributed to the main program. No usage data for sub1, sub2, f1 or f2. What happened?

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.00     18.12    18.12        1    18.12    18.12  MAIN__

Call graph:
index % time    self  children    called     name
               18.12    0.00       1/1           main [2]
[1]    100.0   18.12    0.00       1         MAIN__ [1]
-----------------------------------------------
                                              <spontaneous>
[2]    100.0    0.00   18.12                 main [2]
               18.12    0.00       1/1           MAIN__ [1]
-----------------------------------------------
```

A simple gprof example – Lesson #1

An optimizing compiler may make transformations to your code that render it unrecognizable. In this case, the compiler performed function inlining. To avoid call overhead and to enable further optimizations, the body of the function was moved into the calling routine.

In fact, two levels of inlining were performed

- sub1 and sub2 inlined into main program (shown below)
- f1 and f2 inlined into sub1 and sub2

```
call sub1(x,y,z,n)
call sub1(y,x,z,n)
call sub2(x,y,w,n)
```

```
do i=1,n
  z(i) = f1(x(i),y(i)) + f2(x(i),y(i))
enddo
do i=1,n
  z(i) = f1(y(i),x(i)) + f2(y(i),x(i))
enddo
do i=1,n
  z(i) = f1(x(i),y(i))
enddo
```

A simple gprof example

Let's profile again, this time with function inlining explicitly disabled using -inline-level=0. Things now look more like what we had expected.

```
Flat profile:
%   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 39.90     10.41    10.41 3000000000     0.00     0.00  f1_
 26.04     17.20     6.79 2000000000     0.00     0.00  f2_
 20.57     22.56     5.37        2     2.68     9.55  sub1_
  7.25     24.45     1.89        1     1.89     5.36  sub2_
  6.25     26.08     1.63        1     1.63    26.08  MAIN__

Call graph:
index % time    self  children    called     name
[lines not shown]
 5.37   13.73       2/2           MAIN__ [1]
[3]     73.2    5.37   13.73       2         sub1_ [3]
                6.94    0.00 2000000000/3000000000     f1_ [4]
                6.79    0.00 2000000000/2000000000     f2_ [5]
[lines not shown]
```

A simple gprof example – Lesson #2

There's one small quirk. The time reported for the entire program using the Linux time utility (65 s) does not match what we see in the grof output (26 s). gprof probably did a reasonable job of reporting where time was spent, but **for accurate timings there's no substitute for measuring directly**. This could be done at the level of the entire program or by inserting instrumentation (timers) into your code.

```
Flat profile:
%   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 39.90     10.41    10.41 3000000000     0.00     0.00  f1_
 26.04     17.20     6.79 2000000000     0.00     0.00  f2_
 20.57     22.56     5.37        2     2.68     9.55  sub1_
  7.25     24.45     1.89        1     1.89     5.36  sub2_
  6.25     26.08     1.63        1     1.63    26.08  MAIN__
```

gprof really doesn't add much overhead

Given the inconsistency between the gprof-reported and Linux-report times with function inlining disabled, maybe gprof is adding significant overhead. Testing indicates this is not the case since we see only a modest and reproducible performance hit using gprof.

My hunch is that this is related to the large number of function calls (~3 billion) for this test problem.

Run times by compiler options

	Inlining enabled	Inlining disabled
With -pg	18	65
Without -pg	18	62

Profiling tools: uProf

- Proprietary tool for profiling on AMD hardware
- Compile with -g flag to get symbol information
- Profiling data available in two formats
  - CSV files – human readable and can be opened in spreadsheets
  - SQLite data base – can be imported into AMD uProf GUI
- In addition to time-based profiling can also access performance counters to investigate cache usage, branch prediction and

uProf workflow

Phase	Description
Collect	Running the application program and collect the profile data
Translate	Process the profile data to aggregate and correlate and save them in a DB
Analyze	View and analyze the performance data to identify bottlenecks

uProf: Collect

Profile data generated with AMDuProfCLI-bin tool collect command, where sampling can be done using predefined profiles (see next two slides) or specific events

```
AMDuProfCLI-bin collect --config <sampling> -o <output> a.out
```

This will generate an <output>.caperf file (Linux) or <output>.prd file (Windows) that is used during the translate stage

Predefined profiles for 'collect --config' option

tbp : Time-based Sampling
Use this configuration to identify where programs are spending time.

assess_ext : Assess Performance (Extended)
This configuration has additional events to monitor than the Assess Performance configuration. Use this configuration to get an overall assessment of performance.
[PMU Events: PMCx076, PMCx0C0, PMCx0C2, PMCx0C3, PMCx0AF, PMCx025, PMCx029, PMCx060, PMCx047, PMCx024, PMCx00E, PMCx043]

branch : Investigate Branching
Use this configuration to find poorly predicted branches and near returns.
[PMU Events: PMCx076, PMCx0C0, PMCx0C2, PMCx0C3, PMCx0C4, PMCx0C8, PMCx0C9, PMCx0CA]

data_access : Investigate Data Access
Use this configuration to find data access operations with poor L1 data cache locality and poor DTLB behavior.
[PMU Events: PMCx076, PMCx0C0, PMCx041, PMCx043, PMCx045, PMCx047]

Predefined profiles for 'collect --config' option

nst_access : Investigate Instruction Access
Use this configuration to find instruction fetches with poor L1 instruction cache locality and poor ITLB behavior.
[PMU Events: PMCx076, PMCx0C0, PMCx084, PMCx085]

memory : Cache Analysis
Use this configuration to collect profile data using instruction-based sampling for cache access analysis. Samples are attributed to instructions precisely with IBS.

ibs : Instruction-based Sampling
Use this configuration to collect profile data using instruction-based sampling. Samples are attributed to instructions precisely with IBS.

assess : Assess Performance
Use this configuration to get an overall assessment of performance and to find potential issues for investigation.
[PMU Events: PMCx0C0, PMCx076, PMCx0C2, PMCx0C3, PMCx041, PMCx047]

uProf: A note on paranoid levels

The type of profiling that you are allowed to do depends on the Linux perf_event_paranoid setting. In order to use all of uProf's predefined profiles, we would need to set to a value that introduces security risks

https://unix.stackexchange.com/questions/519070/security-implications-of-changing-perf-event-paranoid

https://stackoverflow.com/questions/51911368/what-restriction-is-perf-event-paranoid-1-actually-putting-on-x86-perf

To balance security and usability, we will be setting to 1, which allows time based and data access based profiling. You should also be able to access specific counters at this level.

uProf: Translate

Translation of the profiling data is done with AMDuProfCLI-bin tool report command

```
AMDuProfCLI-bin report -i <output>.caprof
```

This will generate a new directory named <output>, which contains two files: <output>.csv and <output>.db. The former is plain text, while the latter is a data base file that is read by the GUI

uProf: Analyze

- Profiling data can be analyzed anywhere. My preferred way of doing this would be to install the AMDuProf tool (GUI) on my local machine, but AMD does not currently have an implementation for Mac.
- For now, I'm pasting the output into Excel. This works pretty well although it requires a little formatting (column resizing) and familiarizing yourself with the output.

uProf time based profiling

Let's revisit the same code that we examined with gprof, except using uProf's time based sampling. Note that we compile with -g to get symbol data and got rid of -pg since that's specific to gprof

Build: ifort -g -march=core-avx2 -O3 [-inline-level=0] -o intro intro.f

Collect: AMDuProfCLI-bin collect --config tbp -o intro-tbp ./intro 1000000000

Translate: AMDuProfCLI-bin report -i intro_inline-tbp.caperf

uProf time based profiling – with inlining

uProf assigned usage to lines of source code – in this case the source that resulted after subroutines and functions were inlined. Of course, this will always be imperfect since the optimizing compiler will transform the code.

HOT FUNCTIONS (Sort Event - Timer samples)	
FUNCTION	Timer samples (seconds)
MAIN__	14.287
clear_page_rep	2.128
__do_page_fault	0.572
get_page_from_freelist	0.252
_raw_spin_unlock_irqrestore	0.244
free_unref_page_list	0.166
__handle_mm_fault	0.091
mem_cgroup_throttle_swaprate	0.068
handle_mm_fault	0.053
try_charge	0.048

Line	SourceCode	Timer samples (seconds)
MAIN__		14.287
4	f1 = sqrt(sqrt(x))/sqrt(y) + y/x	4.944
12	f2 = sqrt(sqrt(x*y))/sqrt(y/x) + x/y	7.268
24	do i=1,n	0.532
25	c(i) = f1(a(i),b(i)) + f2(a(i),b(i))	0.55
37	do i=1,n	0.349
38	c(i) = f1(b(i),a(i))	0.264
59	x(i) = i*1.1	0.187
60	y(i) = i/3.3	0.193

uProf time based profiling – without inlining

Profiling without inlining looks like what we saw with gprof, with one important exception – the amount of time assigned to routines is very close to the actual wall clock time. Note also that assignment of time to lines of code is not quite right.

```
HOT FUNCTIONS (Sort Event - Timer samples)
FUNCTION                Timer samples (seconds)
f1_                                  32.645
f2_                                  18.508
sub1_                                5.375
clear_page_rep                       2.168
sub2_                                1.011
__do_page_fault                      0.577
MAIN__                               0.381
get_page_from_freelist               0.263
_raw_spin_unlock_irqrestore          0.238
free_unref_page_list                 0.175
```

```
Line      SourceCode                        Timer samples (seconds)
f1_                                                  32.645
4         f1 = sqrt(sqrt(x))/sqrt(y) + y/x           7.734
6         end function f1                            24.911

f2_                                                  18.508
12        f2 = sqrt(sqrt(x*y))/sqrt(y/x) + x/y       15.976
14        end function f2                            2.532
```

A digression before demonstrating uProf data access profiling

- Although gprof can tell you which functions are taking the most time, it can't provide insights into **why** a particular function is expensive.
- For codes with low computational intensity (i.e., perform relatively small amount of computation for each word of data that is operated upon), performance depends on how well we manage data movement. More specifically, how well we make use of cache.
- uProf gives us access to program counters and lets us go deeper.

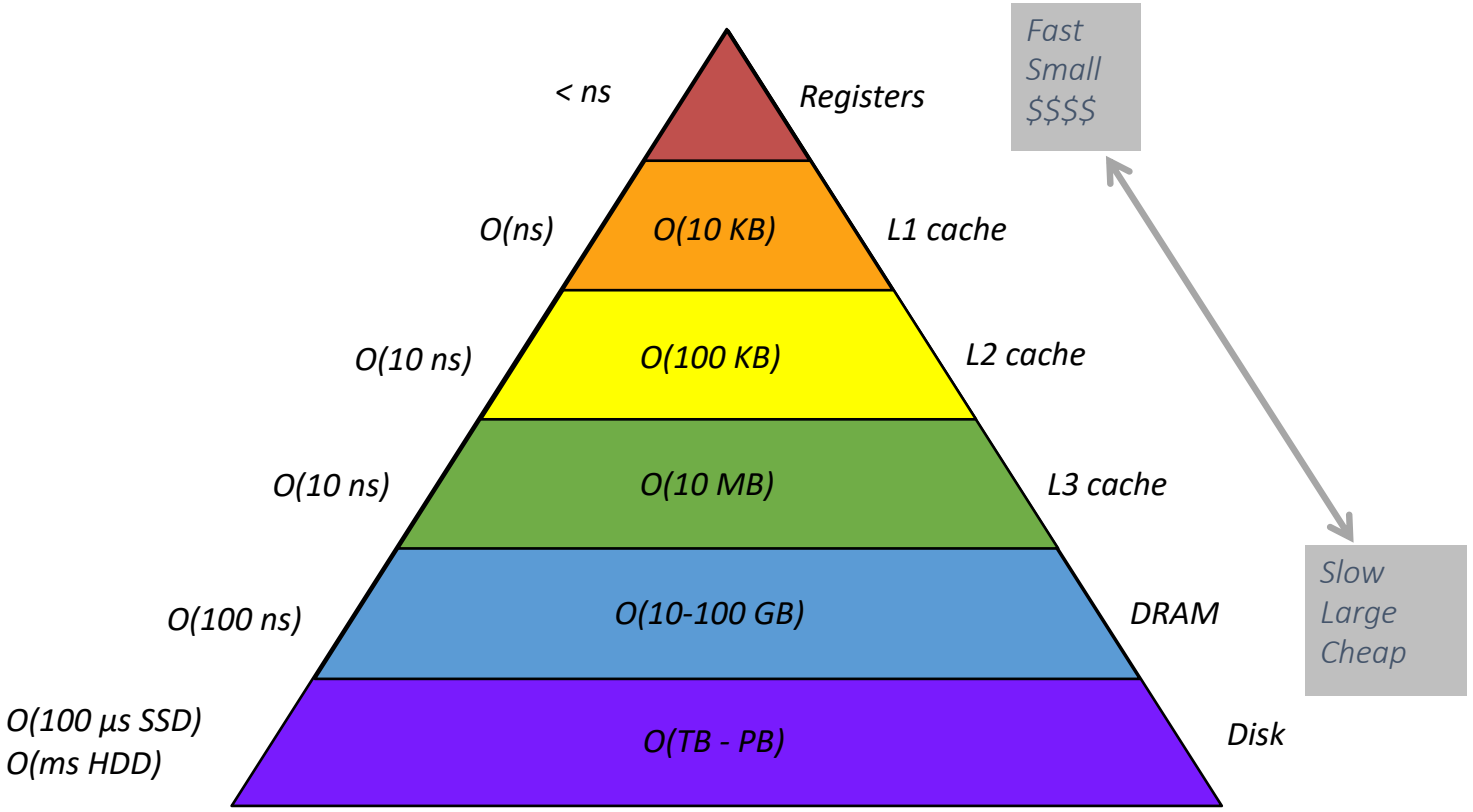
Compute bound vs. memory bound codes

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



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.

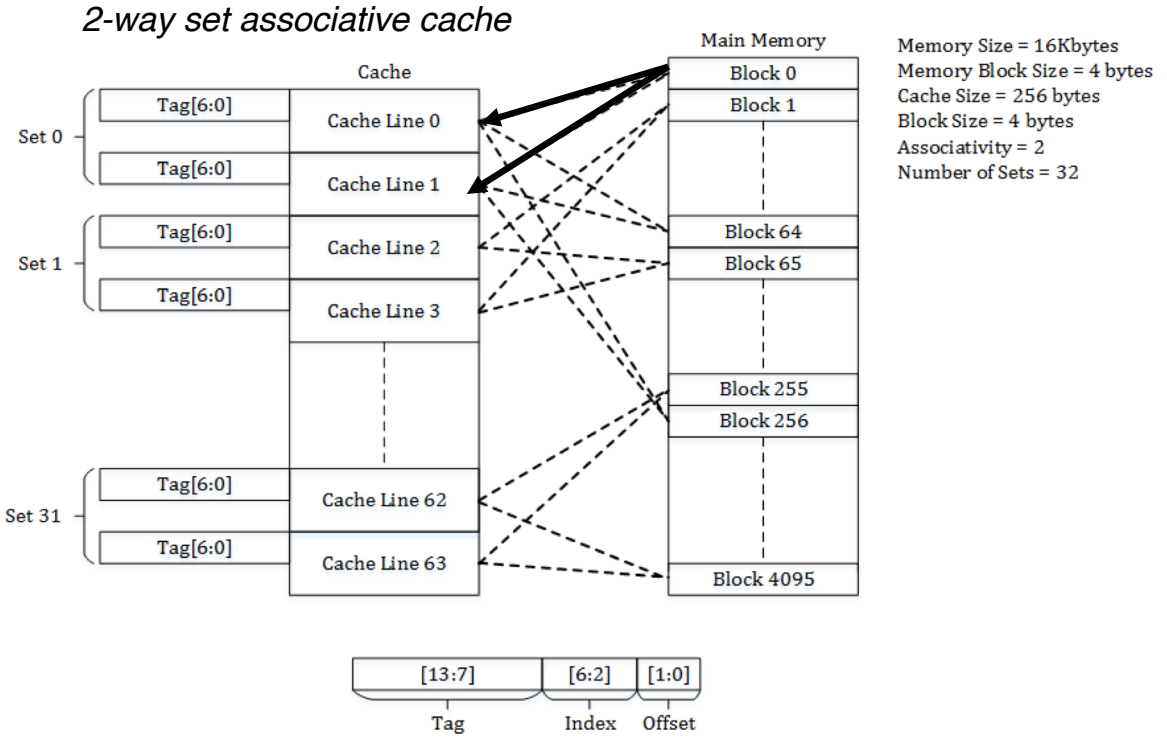
Spatial locality: If a piece of data is accessed, it's likely that neighboring data elements in memory will be needed. To take advantage of spatial locality, cache is organized into lines (typically 64 bytes) and an entire line is loaded at once.

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

Set associative cache

Main memory is much larger than cache and we need a policy for mapping memory locations to cache lines. This is done in predictable way based on the memory address. Most modern CPUs use a set associative cache, where a block of memory maps to multiple cache lines.

Caches also need a replacement policy for discarding to make room for new data. For set associative caches, this can be a variation of least recently used (LRU) or random replacement.

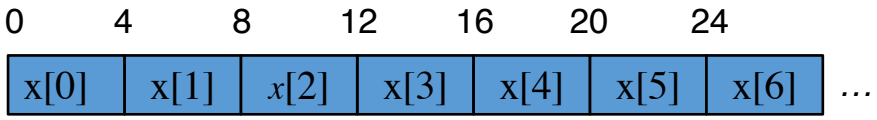


By Snehalc - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=52486741

One-dimensional arrays

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

```
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.

```
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?

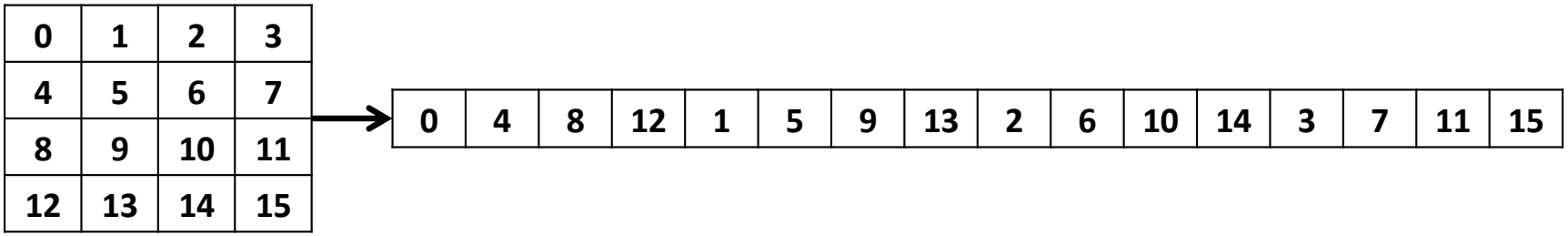
There are no programming constructs that I'm aware of that give you direct control over cache (e.g. load a particular location in memory into cache).

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.

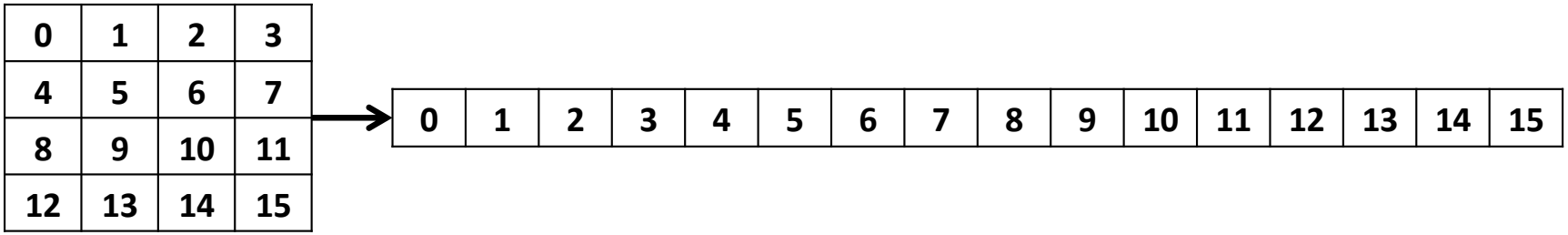
Multidimensional arrays

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



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



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]
   }
}
```

Matrix addition performance

Measuring the run time for the addition of two large NxN matrices reveals some strange behavior. As expected, the performance of the code with the proper loop nesting is much better than that with the improper loops nesting, but the results for N=32,768 are truly puzzling.

Run time for addition of NxN matrices

N	t(s) proper loop nesting	t(s) improper loop nesting
32,767	2.31	29.76
32,768	2.34	**124.39**
32,769	2.32	21.85

Matrix addition performance

Profiling with uProf using the **data_access** sampling configuration provides information on cache behavior. The **Function Detail** section of the report shows that the number of cache misses is significantly higher for the improper loop nesting in general, and is especially large for the N=32,768 problem size

Data cache misses

N	proper loop nesting	improper loop nesting
32,767	10,419	195,533
32,768	11,866	**964,672**
32,769	10,341	116,241

Matrix addition performance

Profiling with uProf using the **data_access** sampling configuration provides information on cache behavior. The **Function Detail** section of the report shows that the number of cache misses is significantly higher for the improper loop nesting in general, and is especially large for the N=32,768 problem size

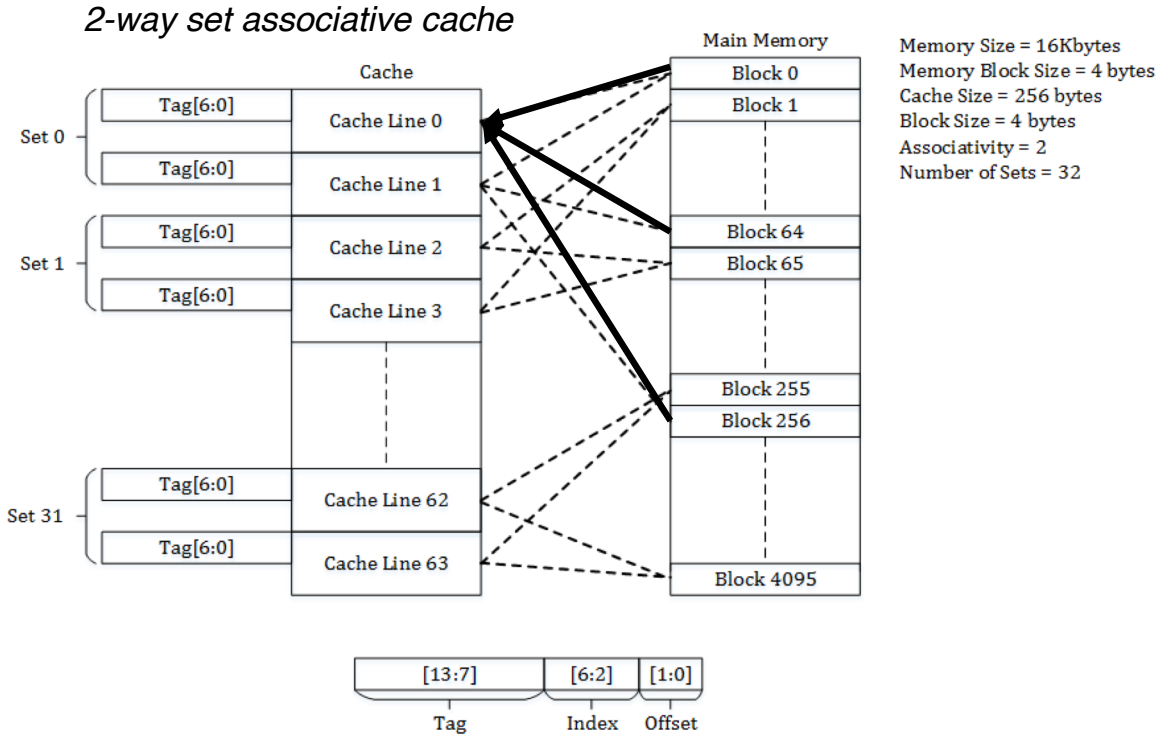
Data cache misses

N	proper loop nesting	improper loop nesting
32,767	10,419	195,533
32,768	11,866	**964,672**
32,769	10,341	116,241

Matrix addition performance

So what's special about N=32,768?

It's a large power of 2 and the mapping of memory to cache is based on a portion of the memory address. When we access the 2d arrays in the wrong order, we're not only making poor use of cache, but we're repeatedly going to the same cache lines. This forces data out of cache before any other words in that line can be used.



By Snehalc - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=52486741

Self-paced exercises – gprof

- Run the gprof example (intro.f code) and convince yourself that you understand the output.
- Modify functions f1 and f2 in intro.f to use sin, cos, and log functions. How does this affect what you see in the gprof output? (Hint – google SVML).
  These functions are expensive and can lead to longer run times – adjust problem size accordingly. For your convenience, commented lines already added to file.
- Break intro.f into multiple files and rerun gprof. What does this tell you about the compiler's inlining capabilities?
  - intro.f → 2 files: main.f, library.f (sub1, sub2, f1, f2)
  - intro.f → 3 files: main.f, subs.f (sub1, sub2), funcs.f (f1, f2)
- Explore other compilers (AOCC flang and GCC gfortran)

Self-paced exercises – uProf

- Read the uProf user guide https://developer.amd.com/wordpress/media/files/AMDuprof_Resources/User_Guide_AMD_uProf_v3.3_GA.pdf
- Once uProf is installed on Expanse
  - Profile intro.f using time based profiling
  - Profile matrix addition (dmadd_[good|bad].f) using time based and data access profiling
  - Look at the effect of omitting -g flag on uProf output
  - Investigate effect of matrix addition problem size. Do you see the same behavior for other powers of two (e.g., N=16,384, 8192) or dimensions that contain large powers of two (e.g., $N=2^{13} \times 3 = 24{,}576$)

Performance Optimization
Tutorial Slides

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.

https://www.nrdc.org/resources/americas-data-centers-consuming-and-wasting-growing-amounts-energy

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.

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

optimization	description
default	varies
-O0	no optimization
-O1	limited optimization
-O2	moderate
-O3	aggressive

compiler	target
AMD clang/flag	-march=znver2
Intel icc/ifort	-march=core-avx2

Optimizing for cache

See slides in first portion of talk for overview of memory hierarchy and optimizing for cache

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.

AOCC (flang)

optimization	proper	improper
default	3.47	9.19
-O0	3.47	9.19
-O1	1.68	5.11
-O2	1.14	4.94
-O3	1.14	4.94

Intel (ifort)

optimization	proper	improper
default	1.19	1.19
-O0	6.67	14.88
-O1	1.25	4.91
-O2	1.19	1.19
-O3	1.19	1.19

AOCC compiler – what happened?

AOCC (flang)

optimization	proper	improper
default	3.47	9.19
-O0	3.47	9.19
-O1	1.68	5.11
-O2	1.14	4.94
-O3	1.14	4.94

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

Intel (ifort)

optimization	proper	improper
default	1.19	1.19
-O0	6.67	14.88
-O1	1.25	4.91
-O2	1.19	1.19
-O3	1.19	1.19.

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)

AOCC (flang)

optimization	proper	improper
default	3.47	9.19
-O0	3.47	9.19
-O1	1.68	5.11
-O2	1.14	4.94
-O3	1.14	4.94

Intel (ifort)

optimization	proper	improper
default	1.19	1.19
-O0	6.67	14.88
-O1	1.25	4.91
-O2	1.19	1.19
-O3	1.19	1.19.

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

```
for (int i=0; i<n; i++) {
  z[i] = x[i]*y[i] + a[i]
}
for (int i=0; i<n; i++) {
  w[i] = x[i] + y[i]*a[i]
}
```

```
for (int i=0; i<n; i++) {
  z[i] = x[i]*y[i] + a[i]
  w[i] = x[i] + y[i]*a[i]
}
```

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.

AOCC (flang)

optimization	fused	unfused
default	6.36	7.03
-O0	6.34	6.95
-O1	3.56	4.54
-O2	2.43	3.24
-O3	2.43	3.24

Intel (ifort)

optimization	fused	unfused
default	2.57	3.32
-O0	7.63	8.26
-O1	2.66	3.47
-O2	2.57	3.31
-O3	2.57	3.32

Loop fission

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

```
for (int i=0; i<n; i++) {
  z[i] = x[i]*y[i] + a[i]
  w[i] = x[i] + y[i]*a[i]
}
```

```
for (int i=0; i<n; i++) {
  z[i] = x[i]*y[i] + a[i]
}
for (int i=0; i<n; i++) {
  w[i] = x[i] + y[i]*a[i]
}
```

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.

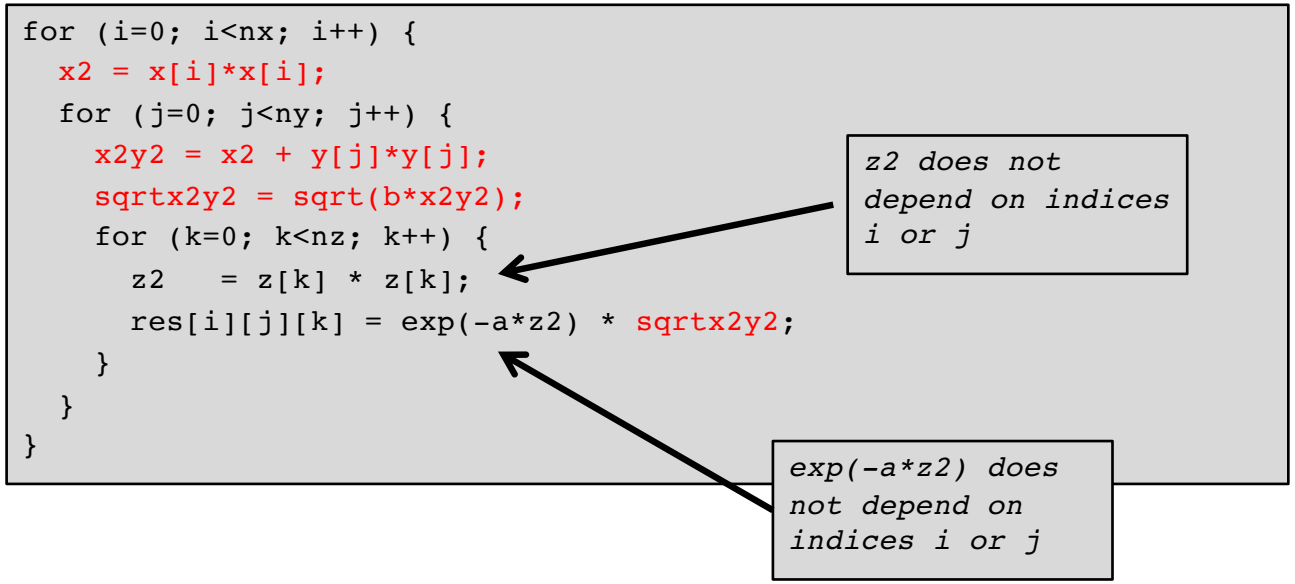
```
for (i=0; i<nx; i++) {
  for (j=0; j<ny; j++) {
    for (k=0; k<nz; k++) {
      x2y2 = x[i]*x[i] + y[j]*y[j];
      z2   = z[k] * z[k];
      res[i][j][k] = exp(-a*z2) * sqrt(b*x2y2);
    }
  }
}
```

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



Loop-invariant code motion – a more complex example

Pre-calculate vector of exp(-a*z2) results and reuse for every set of (i,j). Reduces number of exponential evaluations to nz from nx*ny*nz

```
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

```
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]
}
```

```
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]
}
```

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

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.

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

```
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_miijj(0,i)
   j1=mg63_miijj(1,i)
   j2=mg63_miijj(2,i)
   i0=mg63_miijj(3,i)
   i1=mg63_miijj(4,i)
   i2=mg63_miijj(5,i)
   i3=mg63_miijj(6,i)
   pvi3jj(1 ) = pvi3jj(1 ) + d(i0,i1)*d(i0,i2)*d(i0,i3)*d(i0,j0)*d(i0,j1)
   pvi3jj(2 ) = pvi3jj(2 ) + d(i0,i1)*d(i1,i2)*d(i0,i3)*d(i0,j0)*d(i0,j1)
   pvi3jj(3 ) = pvi3jj(3 ) + d(i0,i1)*d(i1,i2)*d(i1,i3)*d(i0,j0)*d(i0,j1)
   [ Several hundred lines of code not shown ]
   pvi3jj(22) = pvi3jj(22) + d(i0,i2)*d(i0,i3)*d(i0,j0)*d(i1,j0)*d(j0,j1)
   pvi3jj(23) = pvi3jj(23) + d(i1,i2)*d(i0,i3)*d(i0,j0)*d(i1,j0)*d(j0,j1)
   pvi3jj(24) = pvi3jj(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

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

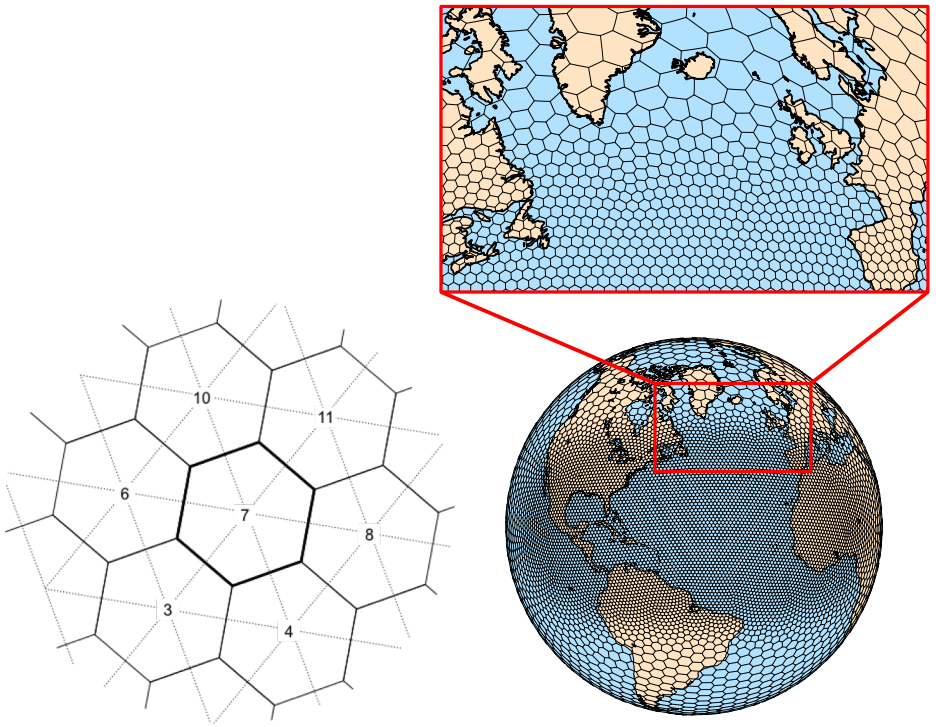
```
for (int i=0; i<n; i++) {
  if (y[i] >= const) {
    break;
  } else {
    // Do stuff
  }
}
```

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] ≤ y[1] ≤ y[2] ≤ y[3] …). 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 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.

```
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)))
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 particular expensive. Look for opportunities to replace with multiplications, particularly if the exponent is known at compile time

pow(x, 8) → x2 = x*x; x4 = x2*x2; x8 = x4*x4

pow(x, 1.5) → y = x * sqrt(x)

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.

sin(x)*cos(x) → 0.5 * sin(2*x)

sin(x)*cos(y)+cos(x)*sin(y) → sin(x+y)

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.

a*sin(x) + b*cos(x) → c = sqrt(a*a + b*b)
phi = atan2(b,a)
c*sin(x+phi)

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 $\text{sqrt}(x^2+y^2) \leq d$

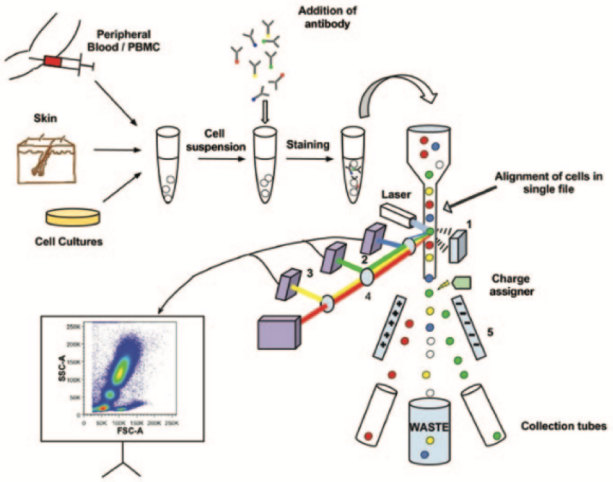
conditions	time
clang w/ force reduction	1.18
clang w/o force reduction	3.20
icc w/ force reduction	1.22
Icc w/o force reduction	3.43

Inter-procedural optimizations

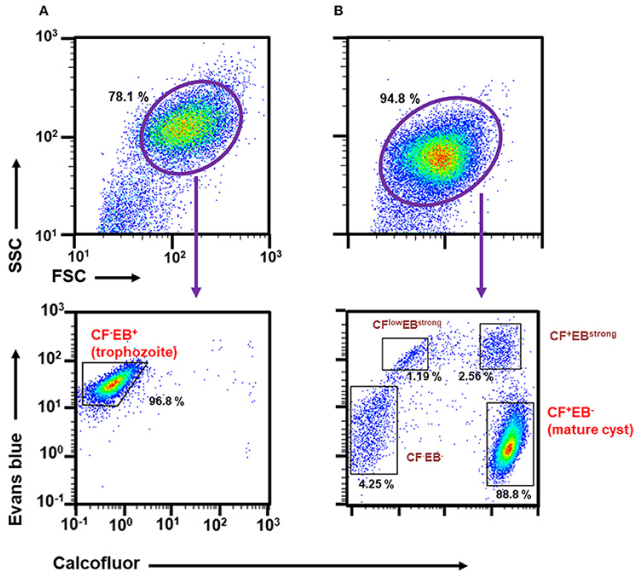
- Inter-procedural optimizations are based on a high-level view of the entire program and span multiple functions
- Compilers are great at optimizing loops (inversion, unrolling, fusion, splitting, peeling, etc.) and statements, but can rarely recognize opportunities for inter-procedural optimizations.
- 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 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.



Jahan-Tigh, et al J. Investigative Dermatology (2012) ***132***



Mi-ichi et al Front. Cell. Infect. Microbiol., 24 July 2018

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

```
Ei = get_avg_dist(rpc[temp_i], temp_i, temp_j, population_ID,
num_real_pop, file_Len, num_dm, norm_data, d1, d2, d3, size_c);

Ej = get_avg_dist(rpc[temp_j], temp_i, temp_j, population_ID,
num_real_pop, file_Len, num_dm, norm_data, d1, d2, d3, size_c);

E1 = get_avg_dist(center_1, temp_i, temp_j, population_ID,
num_real_pop, file_Len, num_dm, norm_data, d1, d2, d3, size_c);

E2 = get_avg_dist(center_2, temp_i, temp_j, population_ID,
num_real_pop, file_Len, num_dm, norm_data, d1, d2, d3, size_c);

E3 = get_avg_dist(center_3, temp_i, temp_j, population_ID,
num_real_pop, file_Len, num_dm, norm_data, d1, d2, d3, size_c);
```

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

```
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.**

Pre-compute results once and reuse

```
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++;
  }
}
```

In function get_avg_dist, eliminated loop and tests on population_ID

```
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