Expanse Webinar – Profiling Tools
AMD uProf

March 18 ,2021

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 (sometimes millions!). 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.

AMD uProf

- Proprietary performance analysis tool for AMD hardware
- Basic time-based profiling (TBP) almost as easy to use as gprof
- 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 instruction based sampling.

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]

PMU = Performance Monitor Unit, PMC = Performance Monitoring Counter

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

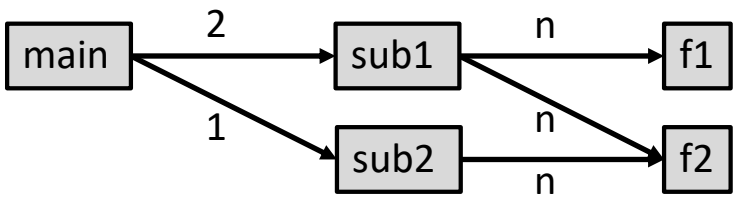
```
AMDuProfCLI-bin report -i <output>.caprof
```

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.

A simple uprof example



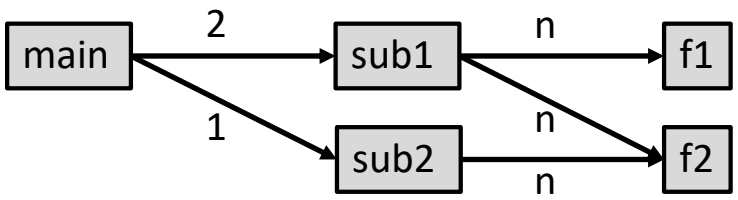
- Main program call sub1 twice and sub2 once.
- sub1 calls f1 and f2 n times
- sub2 calls f1 n times

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

A simple uprof example



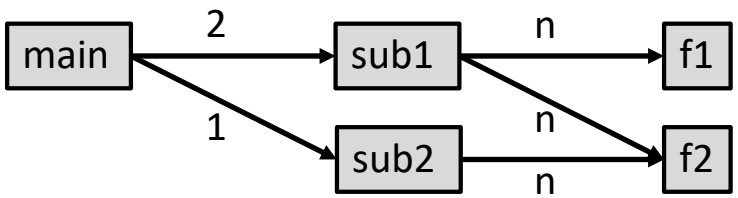
- Main program call sub1 twice and sub2 once.
- sub1 calls f1 and f2 n times
- sub2 calls f1 n times

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

A simple uprof example



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

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

A simple uprof example

```
C     Allocate arrays
      allocate(w(n), x(n), y(n), z(n))

C     Initialize the arrays x and y
      do i=1,n
         x(i) = i*1.1
         y(i) = i/3.3
      enddo

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

```
      subroutine sub1(a,b,c,n)
      do i=1,n
         c(i) = f1(a(i),b(i)) + f2(a(i),b(i))
      enddo
      end subroutine sub1
```

```
      subroutine sub2(a,b,c,n)
      do i=1,n
         c(i) = f1(b(i),a(i))
      enddo
      end subroutine sub2
```

```
      double precision function f1(x,y)
      f1 = sqrt(sqrt(x))/sqrt(y) + y/x
      return
      end function f1
```

```
      double precision function f2(x,y)
      sqrt(sqrt(x*y))/sqrt(y/x) + x/y
      return
      end function f2
```

uProf csv output

```
AMDuProf PROFILE REPORT

EXECUTION
Target Path:,"/home/sinkovit/profiling_tutorial/TimeBasedSampling/intro_inline"
Command Line Arguments:,"1000000000 "
Working Directory:,"/home/sinkovit/profiling_tutorial/TimeBasedSampling"
Environment Variables:
CPU Details:,"Family(0x17), Model(0x31), Number of Cores(128)"
Operating System:,"LinuxCentOS Linux 8 (Core)-64 Kernel:4.18.0-147.el8.x86_64"

PROFILE DETAILS
Profile Session Type:,"Time-based Sampling"
Profile Scope:,"Single Application"
CPU Mask:,"0-127"
CPU Affinity Mask:,"0-127"
Profile Start Time:,"Mar-05-2021_12-18-04"
Profile End Time:,"Mar-05-2021_12-18-24"
Profile Duration:,"20 seconds"
Data Folder:,"/home/sinkovit/profiling_tutorial/TimeBasedSampling"
Virtual Machine:,"No"
Call Stack Sampling:,"False"

Monitored Events:,Name,Interval
,"Timer samples",1000 micro seconds

"HOT FUNCTIONS (Sort Event - Timer samples)"
FUNCTION,"Timer samples" (seconds)
"MAIN__",14.1690
"clear_page_rep",2.1930
"__do_page_fault",0.5280
"get_page_from_freelist",0.2800
"_raw_spin_unlock_irqrestore",0.2240
"free_unref_page_list",0.2020
"__handle_mm_fault",0.1100
"mem_cgroup_throttle_swaprate",0.0620
"handle_mm_fault",0.0560
"mem_cgroup_commit_charge",0.0500


"HOT PROCESSES (Sort Event - Timer samples)"
PROCESS,"Timer samples" (seconds)
/home/sinkovit/profiling_tutorial/TimeBasedSampling/intro_inline (PID - 70949),18.5740

Function Detail Data
Function,File,Line,SourceCode,"Timer samples" (seconds)
"MAIN__",,,,14.1690
,/home/sinkovit/profiling_tutorial/TimeBasedSampling/intro.f,4,"f1 = sqrt(sqrt(x))/sqrt(y) + y/x",5.0690
,,12,"f2 = sqrt(sqrt(x*y))/sqrt(y/x) + x/y",7.0880
,,24,"do i=1,n",0.5200
,,25,"c(i) = f1(a(i),b(i)) + f2(a(i),b(i))",0.5200
,,37,"do i=1,n",0.3100
,,38,"c(i) = f1(b(i),a(i))",0.2850
,,59,"x(i) = i*1.1",0.2010
,,60,"y(i) = i/3.3",0.1760
```

Execution environment and profile details

Hot functions

Function details

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

Assignment of time to lines of code is not quite right. In particular, function f1 assigns most of the time to `end function f1` rather than the statement that that does the real work

```
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 performance counters and lets us go deeper.

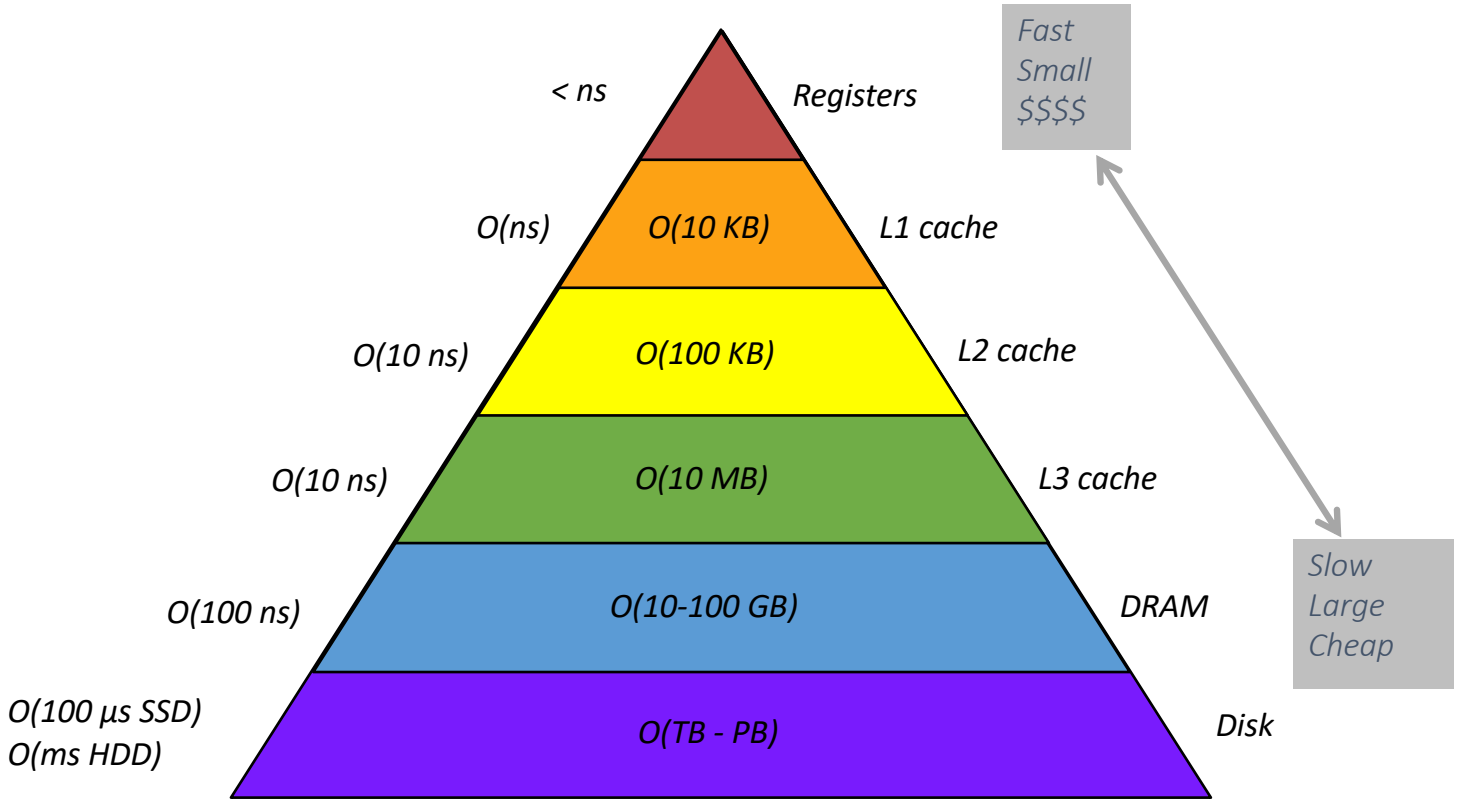
Compute bound vs. memory bound codes

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.

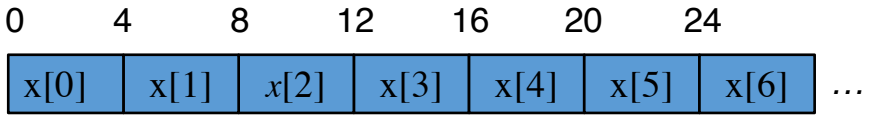
Spatial locality: If a piece of data is accessed, it's likely that neighboring data elements in memory will be needed. To take advantage of spatial locality, cache is organized into lines (typically 64 bytes) and an entire line is loaded at once.

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

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?

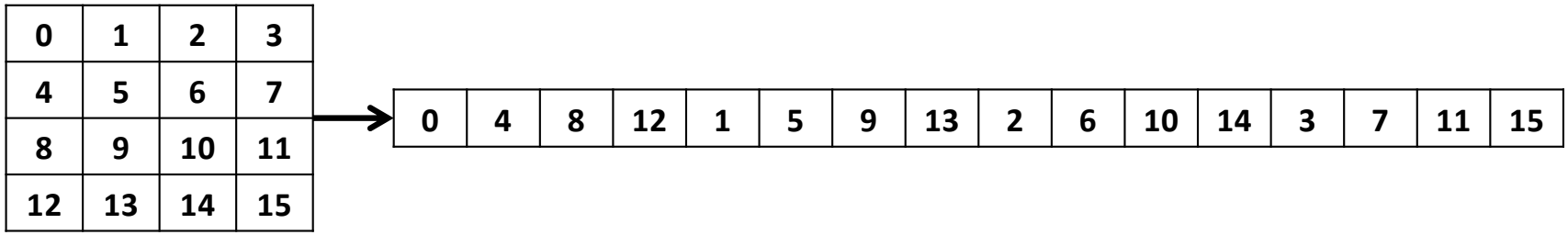
There are no programming constructs that I'm aware of that give you direct control over cache (e.g. load a particular location in memory into cache).

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.

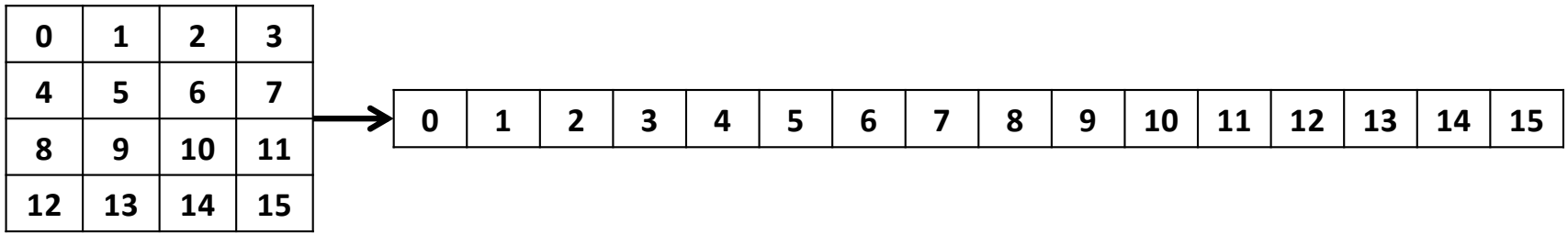
Multidimensional arrays

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 data access profiling

There's no need to do time based profiling with this code – it's just one routine and we already know where the time is being spent. Instead, we'll do data access profiling to get a deeper understanding.

Build:	gfortran -march=znver2 -g -O3 -o dmadd_bad dmadd_bad.f
Collect:	AMDuProfCLI-bin collect --config data_access -o dmadd_bad-da ./dmadd_bad 32768
Translate:	AMDuProfCLI-bin report -i dmadd_bad-da.caperf

Build:	gfortran -march=znver2 -g -O3 -o dmadd_good dmadd_good.f
Collect:	AMDuProfCLI-bin collect --config data_access -o dmadd_good-da ./dmadd_good 32768
Translate:	AMDuProfCLI-bin report -i dmadd_good-da.caperf

Matrix addition cache misses

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

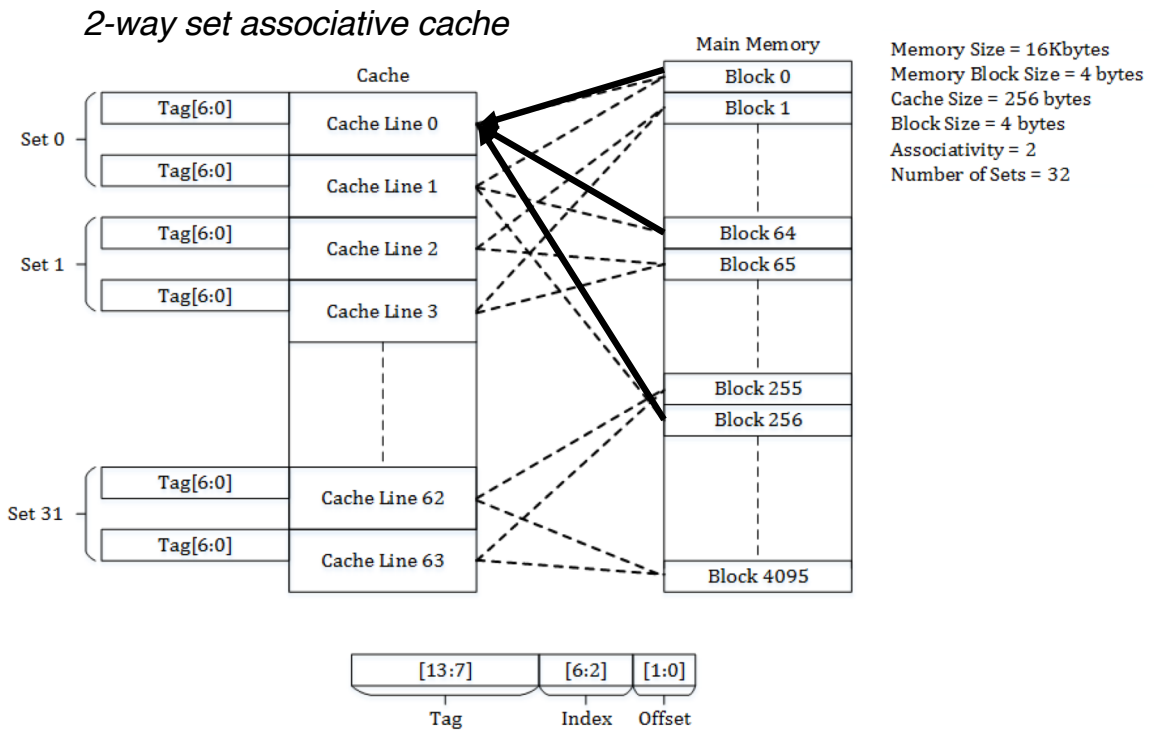
Data cache misses

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 cache usage

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.

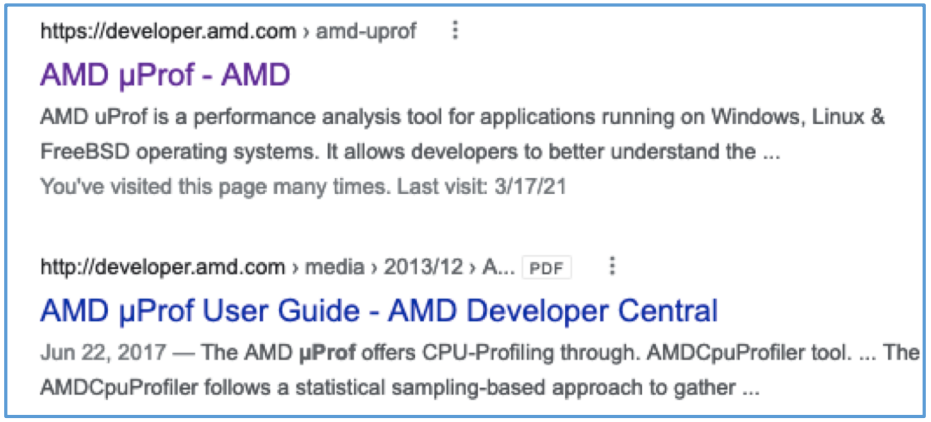


By Snehalc - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=52486741

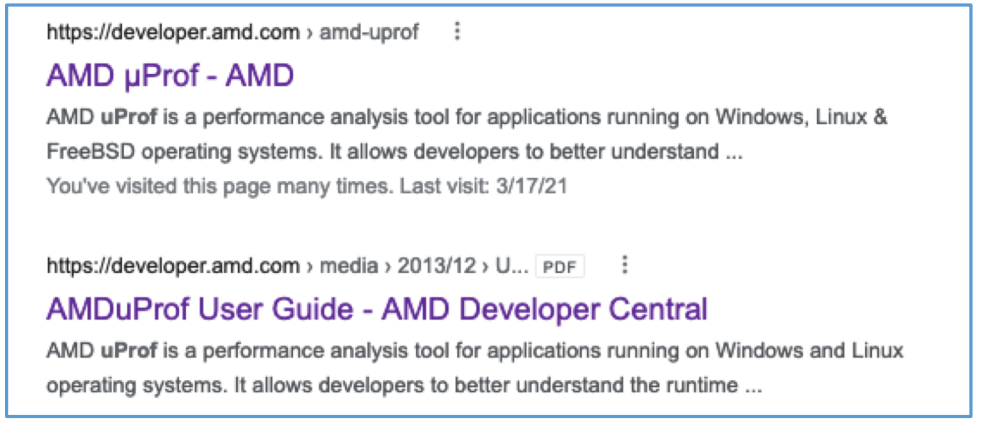
Where to go for help – uProf or µProf?

AMD uses the two names interchangeably, sometimes on the same web page or document. Google searches using “uProf” and “µProf” seem to yield the same hits for the official AMD pages, but different hits for non-AMD pages

µProf (top 2)



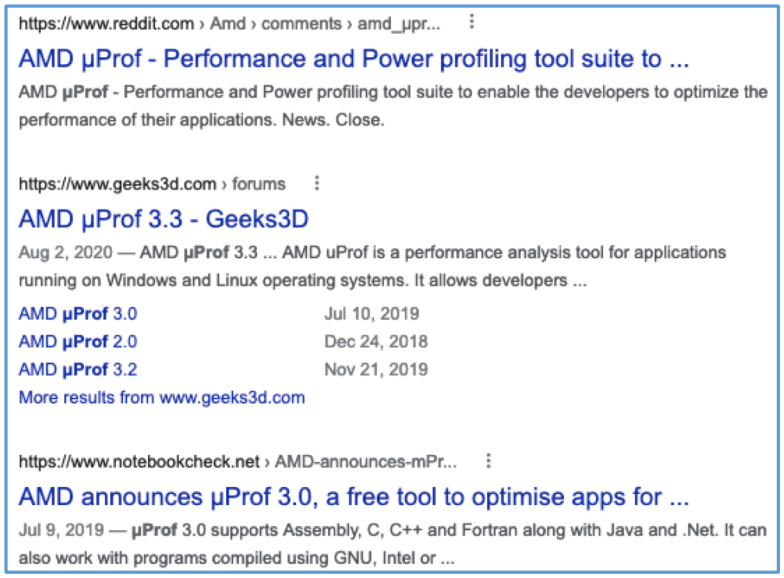
uProf (top 2)



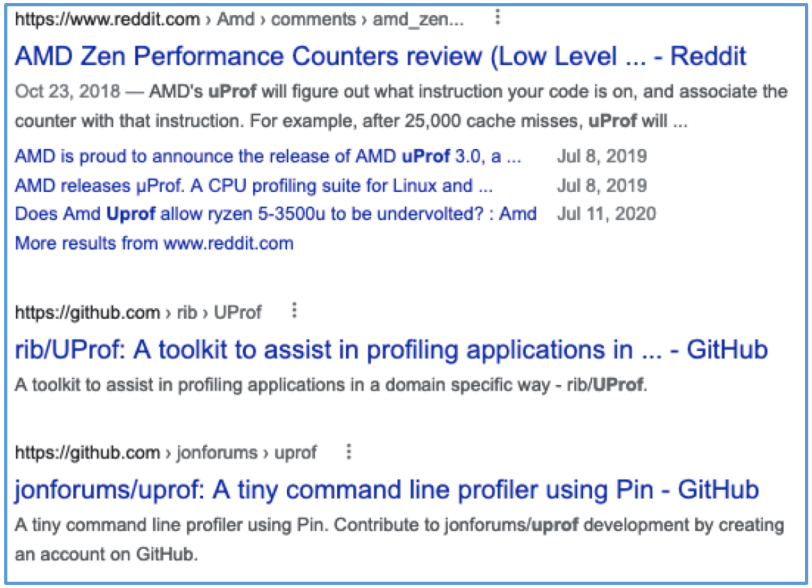
Where to go for help – uProf or µProf?

uProf seems to bring up more unrelated hits, but you'll probably want to do searches using both to track down community input.

µProf (next 3)



uProf (next 3)



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 (matrix addition example)
  - Profile matrix addition (dmadd_[good|bad].f) using data_access
  - Look at the effect of omitting -g flag on uProf output
  - Investigate effect of 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., $N=2^{13} \times 3 = 24,576$)
  - Explore other compilers (AOCC flang and GCC gfortran)

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

Self-paced exercises – uProf

- Once uProf is installed on Expanse (intro example)
  - Profile intro.f using time based profiling (tbp)
  - Look at the effect of omitting -g flag on uProf 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 uProf output? (Hint – google SVML).
  - Break intro.f into multiple files and reprofile. 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)

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