Lab 1 — locality in a CPU

Out: 1/26/2021 Due:2/2/2021, 2/9/2021, 2/16/2021

In this lab you will use dense matrix-matrix multiplication to study the effects of locality on performance and power dissipation. Matrix multiplication is deceptively simple yet actually offers many opportunities for optimization. You will also learn about 3 tools that can help with architecture optimization and research (performance counters, CACTI, and Pin).

You can use just about any computer to run this lab and instructions are provided for Linux machines. However, for consistency, we would like you to use one of the systems hosted by TACC. We have an allocation that will allow you to explore and we will add you to that allocation (a separate "homework" for creating a TACC account will be released as well). If you're curious, you may enjoy comparing TACC to your own machine. When you compile your code, please use reasonable compiler optimization options. I suggest sticking to gcc and using -O2, if you want to experiment with other options (https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Don't forget to report which flags you use and don't forget that you learn a lot more by optimizing yourself when possible. You may also find it instructive to experiment with other compilers (llvm or, if using TACC machines icc). When comparing optimizations take a look at what's actually produced — especially if any SIMD is being produced. Understanding the interactions between hardware and software, often requires understanding software at its lowest level.

IMPORTANT: The lab has three parts with different due dates. Each part will be graded separately, though because of the large number of students and small amount of TA support, do not expect feedback between lab parts. From prior experience this lab takes more time than you might expect and you should start early. I recommend you read the whole lab instructions first (not just one part a time). If you miss the due date, you may submit until Friday of the same week with a 10% penalty and until Monday of the following week with a 25% penalty. Later submissions will not be accepted.

Emphasis and grading

This lab is about locality and matrix multiplication is used as a teaching example. The optimization possibilities for this application are almost endless, and it is not the intent that you try to explore them all. Rather, try to focus on those that are directly related to the learning goals of this class. Basically, effort placed in optimizations that are not locality optimizations will be rewarded less (in grading) than ideas related to locality, methodology, or tools introduced in this lab (it still will be rewarded though, of course).

Grading will address *all* the specific questions that appear below as well as any additional comments or ideas you share.

We will grade using a 5-level scale (I expect the final class grades to be in the A — B range with more 'A's and 'A-'s than 'B's):

5	Truly remarkable work
4	Exceeded expectations (specifically with regards to learning goals)
3	Met expectations (which are high)
2	Did not meet all learning goals expected
1	Requires significant changes

Matrix multiplication

The following C code is the entire matrix multiplication program! We will only use the simple $N3$ algorithm in this lab.

```
1.  #include <stdlib.h>
2.  // define the matrix dimensions A is MxP, B is PxN, and C is MxN
3.  #define M 512
4.  #define N 512
5.  #define P 512
6.  // calculate C = AxB
7.  void matmul(float **A, float **B, float **C) {
8.   float sum;
9.   int  i;
10.  int  j;
11.  int  k;
12.
13.  for (i=0; i<M; i++) {
14.   // for each row of C
15.   for (j=0; j<N; j++) {
16.    // for each column of C
17.    sum = 0.0f; // temporary value
18.    for (k=0; k<P; k++) {
19.     // dot product of row from A and column from B
20.     sum += A[i][k]*B[k][j];
21.    }
22.    C[i][j] = sum;
23.   }
24.  }
25. }
26. // function to allocate a matrix on the heap
27. // creates an mXn matrix and returns the pointer.
28. //
29. // the matrices are in row-major order.
30. void create_matrix(float*** A, int m, int n) {
31.  float **T = 0;
32.  int i;
33.
34.  T = (float**)malloc( m*sizeof(float*));
35.  for ( i=0; i<m; i++ ) {
36.    T[i] = (float*)malloc(n*sizeof(float));
37.  }
38.  *A = T;
39. }
40.
41. int main() {
42.  float** A;
43.  float** B;
44.  float** C;
45.
46.  create_matrix(&A, M, P);
47.  create_matrix(&B, P, N);
48.  create_matrix(&C, M, N);
49.   // assume some initialization of A and B
50.  // think of this as a library where A and B are
51.  // inputs in row-major format, and C is an output
52.  // in row-major.
53.   matmul(A, B, C);
54.
55.  return (0);
56. }
```

[$[Get Code]]

Part 1 — Analytical Modeling (Due 2/5/2021)

In the next two parts you will look at 2 of the 4 most common ways of doing architecture research: measurements on real machines where we have limited or no control over parameters, and simulation, which can be slow and does not scale well. In this part we'll look at analytical modeling, which can be very useful to evaluate trends and look at scaling beyond the capabilities of simulators. More importantly it lets you roughly verify that what you are measuring makes sense — which is very important. The final common technique is prototyping, by the way.

Question 1

Derive formulae for the locality of the original (triply-nested ijk loop), cache-aware (single-level tiled), and cache-oblivious implementations for one level of locality. The formulae should represent what fraction of accesses (access = operand read or write) are serviced by a local store of a given capacity. Assume the matrix is much larger than the storage capacity. you can think of this store as an ideal cache (i.e., LRU replacement, fully associative, one word per cache line).

Draw a graph of the locality of each of the three implementations as the amount of storage grows.

If you get to this part before we cover this in class, please look at the following presentations from UIUC and Bryant and O'Hallaron for cache-aware matmul and the papers linked here.

Question 2

Extend your model to a storage hierarchy (e.g., registers, L1, L2, memory). Plot some results to gain some intuition.

Question 3

An important architecture trend is a move toward heterogeneous memory systems that include both conventional off-package DRAM in addition to higher-latency and lower-bandwidth non-volatile memory and/or higher-bandwidth on-package DRAM. How would you extend the model of Question 2 to such a heterogeneous-memory system? For example, consider what relative values of bandwidth and latency have a significant impact on the performance of matrix multiplication, what capacities make sense, and what level of performance the processor has.

Part 2 — Optimizing Performance (Due 2/11/2021)

In the second part of the lab you will try to improve the performance of the baseline version above on a uni-processor. Most of the performance to be gained is with locality optimizations, followed by converting the code to use modern processor's short-vector SIMD units. We will focus on the locality part (properly-written code will be auto-vectorized by the compiler, which you can test).

In order to improve performance we will need a good way to define and measure performance. We will use '**GFLOPS**' (giga-FLOPS, billions of floating-point operations per second) as our measure of performance. To calculate the GFLOPS of our application we will need to measure the number of operations executed as well as the time spent in the calculation.

Measuring the number of operations in this case is very simple. All the computation occurs in the loop nest starting on line 15 and includes one addition and one multiplication in the inner-most loop, so the total number of computations is $2 \times M \times N \times P$. In general it is important to count the actual number of operations *required by the algorithm* when measuring performance and make sure not to include operations that were added as part of coding or optimization.

Measuring time is a bit more tricky because we cannot simply use the built-in OS time measurement because of accuracy issues. To allow more accurate measurements processors provide *performance counters* in the hardware that measure various events, such as cycles, instructions retired, branches, cache accesses, … We will use the performance counters to measure the number of cycles required by our application. Plus, you should report the statistics of performance counters collected from hardware and matmul application using tools. Due to run-to-run variance, you should run enough times to get average, standard deviation and relative standard deviation.

There are many measurement packages that ease the use of hardware performance counters. One of the best and best supported is PAPI, developed at the University of Tennessee Knoxville. Unfortunately, using PAPI requires a kernel patch to Linux, so we will use the perf tool instead. Tutorial and examples for **perf** usage are available at perf wiki and perf example. We also give some simple usage examples in the steps below, but we suggest reading the tutorial to figure out what is possible with perf. The lab will give you a taste of how to measure performance and the questions will point out some potential pitfalls. If you have access to PAPI and know, or want to learn, how to use it — feel free to do so.

> If you are particularly interested in the topic of measurement, I strongly suggest **Whaley, R.C. and Castaldo, A.M.***Achieving accurate and context-sensitive timing for code optimization.*. ((URL)) as a very practical and to-the-point paper.

Step 1: Setup

You will measure performance for matrix multiplication on the Stampede2 system available in TACC. This step 0 is for those who are not familiar with using TACC resources and you can skip this if you are already familiar with it.

First, there are two ways to log in to TACC, one with a TACC Account or another with your UT EID. Either way should work but encourage you to login with your UT EID so that we can find you to assign and manage allocation. Once you are done with your account, then you can access Stampede2 with ssh from your local machine.

There are compute nodes and login nodes. You log in to a login node first, but remember that the login nodes are not meant for any compute-intensive job. You may only use a login node to edit source code, explore directories, submit your jobs to compute nodes, or do very lightweight jobs such as simple compilation. However for measurement, you need to access a compute node, which will be explained later. Note that there are two types of compute nodes on Stampede2. We will be using the Skylake (SKX) nodes.

You are encouraged to understand the available filesystems: $HOME, $WORK, and $SCRATCH. Notice that your $HOME directory has very limited capacity, 10GB. If you need more storage, then you can use $WORK, which should be enough for this assignment. In order to navigate the shared file system, please refer to "Managing Your Files".

Finally, in order to access the compute nodes, there are 3 ways: submit a batch job, interactive session, and interactive session using ssh. We encourage you to use the first method to access the compute node because you can easily waste the limited compute time with interactive sessions if you happen to forget to exit from the compute node after you are done with your experiments. If you have some jobs to evaluate, then you may consider to submit a batch job with sbatch, or if you are still working on implementations and debugging, and thus like to see immediate results in an interactive way, then an interactive session with idev would be a good option. Once you get the compute node allocated, you are the only person who uses that node for the allocated node hours. Please check the node hour usage regularly since we have very limited node hours shared by the entire class. If you happen to submit jobs by mistakes,then you can cancel with scancel to save our limited allocations. To monitor your submitted jobs to the compute nodes, you can use squeue. Last but not least, you need to make sure which compute node you'd like to access and run your jobs on, which is called Slurm Partitions or Queue. You will need to specify the queue when you try to access compute nodes in any ways discussed above. Development queues have shorter max duration of 2 hours than normal queues. Development queues are intended to be accessed with lower wait times, but that is not always the case.

The above is the minimum for you to use Stampede2. For more interested and patient students, you may want to take a look at the entire user guide. For example, one may be interested in trying other tools for performance monitoring in addition to perf. Then you can find more information about what's already available (or pre-installed) on Stampede2 in "Using Modules". There you could find **PAPI** is already available as a module or Intel Vtune Amplifier as well.

Lastly, if you run independent multiple jobs, which are not measured for performance, utilizing multiple cores after allocating a compute node (there are 48 cores in a SKX node), please use launcher to run multiple jobs with a single batch instead of batching each run at a time. This will save a lot of node hours! Check this sample launcher script.

Please pay attention to the time you spend on TACC machines. It's a shared and valuable resource that we do not want to abuse.

Step 2: Basic MatMul

Stampede2 already supports the perf command line tool. If you'd like to try perf on your own machines, then you can install the perf package by running **apt-get install linux-tools-common** and **apt-get install linux-tools** if you're using Ubuntu. If you're using Fedora, **yum install perf** will work.

The following command shows an example of measuring user-level events (assuming your executable binary is named matmul):

perf stat -e [event,...] COMMAND [ARGS]

(Example usage) perf stat -e cycles:u,instructions:u,cache-references:u,cache-misses:u ./matmul

(Some useful events)

- cycles:u : Total cycles (Be careful of what happens during CPU frequency scaling with respect to performance)
- instructions:u : Retired instructions
- cache-references:u : Usually, indicates LLC accesses, and it may include prefetch, coherence message depending on your CPU
- cache-misses:u : Usually, indicates LLC misses
- L1-dcache-load:u : L1 cache load accesses
- L1-dcache-load-misses:u : L1 cache load misses
- L1-dcache-stores:u : L1 cache store accesses
- L1-dcache-store-misses:u : L1 cache store misses

The postfix **:u** is for measuring performance counters only for user-level instructions (default is measuring at both kernel and user levels).

You can add more monitored events in a single perf command, and some events are useful to understand and explain the differences between measurements and code versions (some of them are listed above).

The command **perf list** (or we recommend to use pmu-tool's **ocperf.py list**) shows the events that can be counted in the compute node (SKX) (please note that some of the listed events are not actually available in your CPU, though). You can find more details of each event at perf_event_open() documentation, but please note that exact meaning of each event may vary depending on your CPU. More perf examples can be found in Brendan Gregg's blog.

When you increase the number of monitored events, however, you need to be careful because there is a limited number of hardware performance counters. According to perf wiki, when monitoring more events than there are available hardware performance counters, events are multiplexed over the counter registers and this may decrease accuracy. For example, the Intel 4-core i5–2520M 2.5GHz CPU has 11 hardware performance counter registers total. I recommend using fewer than 5 events simultaneously. If you use too many events in a single perf command, some events will not generate useful numbers (often registering a zero).

Due to dynamic frequency scaling, care must be taken when you convert a cycle count to execution time. One option is to fix CPU frequency of your CPU with **cpufrequtils**, but we are not considering it in this lab.

Note that your matrix multiplication software runs not only matmul(), but also create_matrix() and other functions, so estimated GFLOPS may be a bit off, but still useful to evaluate performance of your software with different configurations and various optimization techniques. Please measure the performance of your application at least 20 times.

Alternatively, we can measure GFLOPs with toplev.py tool in pmu-tools. The toplev.py script performs a "top-down analysis" and provides a performance breakdown using performance counters. For the detailed information about the top-down analysis, please take a look at: "A Top-Down Method for Performance Analysis and Counters Architecture". There are more tools which are useful for profiling performance with performance counters. ocperf.py, for example, is a wrapper for perf that provides a full core performance counter event list for common Intel CPUs while perf sometimes requires you to set raw events to MSR register which is cumbersome. If you find any event which is not available from perf list, then you may have to consider trying "ocperf.py list" or papi_avail to search for more available counters (but not ported to perf).

Question 4

How many GFLOPS did the basic version achieve on your test machine? Please use three different matrix sizes: M=N=P=32, M=N=P=512, and M=N=P=4096.

Things to address: What machine did you use? Why are several measurements taken? Which one(s) should you report and why (please see the cache performance as well)? How did you convert from cycles to seconds and why did you do it that way? How do you define and measure FLOPs (floating point operations) and do your assumptions match performance counter measurements?

Was there any significant difference between the two matrix sizes in terms of the trends in the timings and cache misses of each measurement? Can you make them behave the same way with regards to timing variance? Should you do it? Please include any code you used. Try measuring performance counters of interest and collecting statistics of performance profile.

Step 3: Optimized MatMul

Now try to improve locality and hence performance. Please think of optimizing registers as well as the memory hierarchy. Try both cache-aware and cache-oblivious methods. You may want to consider larger datasets for this step of the lab to test your ideas. Try measuring cache performance of matmul application using hardware counters.

The A, B, and C matrix formats are all row-major (even though they are just malloced for simplicity (think of this as a library interface). Anything you do to manipulate A and B should be counted for time and accesses (but not count towards instruction count for performance).

Question 5

What techniques that you used worked well and what didn't make much of a difference? Why? Please discuss how your optimization changes number of cache misses in each level, and how it affects overall performance. Please address the different locality mechanisms in hardware when answering this question and document failed and successful techniques (only correct ones of course).

Part 3 — Analyzing Power (Due 2/18/2021)

As architects it is important for us to understand both the software costs (performance) and the hardware costs. Hardware costs are typically categorized as die area, complexity and power/energy. There are other costs to consider such as reliability, features, and so forth that are usually set by the usage model. We will only look at power/energy in this lab.

We discussed the effects locality has on power and energy in class. We will not attempt to develop a power model based on "first principles" and will instead use the well-regarded CACTI tool from HP Labs. CACTI models the hardware costs of caches, SRAMs, and DRAM and the latest version of CACTI (v6.5) can be downloaded at http://www.hpl.hp.com/research/cacti/cacti65.tgz. CACTI also has a web interface, but the web interface consistently uses and out-of-date version of CACTI and is also not as configurable.

To estimate the power and energy consumed by the matrix multiplication application we need to know both how much energy each access to a locality level takes and how many accesses are actually made to each level. To determine this we could use the hardware performance counters to measure cache accesses, but then we would not be able to explore the behavior as we change cache sizes. Therefore, we will use a simple cache simulator built on top of PIN, which is a dynamic hardware instrumentation tool.

We will also measure the energy consumption to run matrix multiplication using perf. Idling resources still consume power. To identify the energy consumed for matrix multiplication, we will first measure power while idling. Idle periods still consume huge amounts of energy! Then, we will measure the energy while running matrix multiplication.

Step 1: Simulate impact of optimizations

Download Pin from Intel here and untar it (you can use /tmp or $SCRATCH if you don't have much quota left). The source/tools/SimpleExamples directory contains dcache.cpp, which implements a one-level data cache. We want to have a 2-level cache model with **LRU replacement policy**, which you can easily create by some artful copying and pasting. This is a good opportunity to learn how to modify existing tools to achieve your goals. Make sure you have some way of changing the cache parameters and sizes (you can just replicate the parameters of the L1 cache). For the cache model, please assume both caches are read- and write-allocate, write-back, and inclusive (this means allocate a line on any read or write miss, allocate in L2 on an L1 miss, and assume that a write-back from L1 never misses in L2 and doesn't change its LRU). Note that the PIN tool will not work on some of the Linux distributions. The message you will get is this error message: "Can not load shared library libm.so.6" . Refer to the release note for tested configurations. You can find more information about Pin and Pin tool in Pin Tutorial.

Once you have the two level model, verify it by running the following steps from within the source/tools/SimpleExamples directory. You will have to change obj-ia32 to obj-ia64 or obj-intel64 depending on the architecture of the CPU you are using. Set the L1 parameters to (size=32KB, associativity=4, block size=32) and the L2 to (size=2048KB, associativity=16, block size=64).

```
make
wget "http://lph.ece.utexas.edu/class/Sp15EE382N/Lab1?action=download&upname=test_cache.c.txt"
mv Lab1\?action\=download\&upname\=test_cache.c.txt test_cache.c
wget http://lph.ece.utexas.edu/downloads/EE382V_Principles/test_cache.out
gcc -O0 test_cache.c -o test_cache
../../../pin -t obj-intel64/dcache.so -- ./test_cache
diff dcache.out test_cache.out
```

[$[Get Code]]

You’re now ready to measure the locality in your matrix multiplication code. Just run you PIN dcache tool as for test_cache, but use your matrix multiplication program instead.

Question 6

Please fill in the table below with the number of references made to each locality level. For registers, just look at the number of accesses required for the actual computation, which you can calculate based on the code. Please use the exact implementations you did in the previous steps. For all the runs, please set the L1 associativity to 8 and the L2 associativity to 16. Some of these runs may take an hour or so.

(Update) Please use 64B cache block size for both L1 and L2. L1 associativity is 8, and L2 associativity is 16.

N=M=P	L1 Size	L2 Size	L1 Tile	L2 Tile	Original				Cache Aware				Cache Oblivious			
	Your processor config		Tile size		Regs	L1	L2	MEM	Regs	L1	L2	MEM	Regs	L1	L2	MEM
4096	32	1024	64	128												
512	8	256	32	64												
512	1	128	16	64												

Why do you think the results differ between the configurations and between the cache-aware and cache-oblivious implementation? Can you make the cache-aware and cache-oblivious behave similarly? If yes, please fill in another table for the new version(s).

Question 7

Now, optimize the matrix multiplication with each input size given cache parameters. “Original” and “Cache Oblivious” can be copied and pasted from the Q6.

(Update) Please use 64B cache block size for both L1 and L2. L1 associativity is 8, and L2 associativity is 16.

N=M=P	L1 Size	L2 Size	L1 Tile	L2 Tile	Original				Cache Aware				Cache Oblivious			
	Your processor config		Tile size		Regs	L1	L2	MEM	Regs	L1	L2	MEM	Regs	L1	L2	MEM
4096	32	1024														
512	8	256														
512	1	128														

Step 2: Performance counter measured impact of optimizations

Question 8

Please fill in the table below with the number of references measured at each locality level. For registers, just use the number in the question 6. Note that the cache associativity is different at each level. Fill in the L1/L2/L3 size of the processor you use (which should be the TACC SKX nodes).

N=M=P	L1 Size	L2 Size	L3 Size	L1 Tile	L2/L3 Tile	Original					Cache Aware					Cache Oblivious				
	SKX 8160			Tile size		Regs	L1	L2	L3	MEM	Regs	L1	L2	L3	MEM	Regs	L1	L2	L3	MEM
4096																				
512																				

Question 9

Compare the access statistics to the memory hierarchy from the model with the measured hardware counters. Why do you think the numbers are different? Suggest multiple reasons and discuss.

Step 3: Using the CACTI Cache Model

Question 10

In this step we will use CACTI to estimate the potential power savings of locality optimizations. Use CACTI to fill in the table below.

For registers, copy the configuration file cache.cfg to a new file called register.cfg, and edit it to represent a "cache" with 128 entries of 32 bits each (this is not accurate, but gives a good sense of things). Set '-cache type' to 'ram' with 1 input and one output port at 350K and 1 bank (also choose ITRS-HP, Conservative, and semi-global). Use 45 as the technology, which is similar to the 45nm technology in use today.

For caches, use cache.cfg, and configure it appropriately. Set '-cache type' to 'cache' with 1 read-write port at 350K and 1 bank (also choose ITRS-HP, Conservative, and semi-global). Use 45 as the technology, which is similar to the 45nm technology in use today.

For DRAM, use dram.cfg, and set size to 512MB, one read/write port, commodity DRAM rather than ITRS-HP, number of bits read 128, and global wires.

	Size	Ways	Linesize	Access Energy
Your L1				
Your L2				
Registers	512	1	4	
L1	32K	8	64	
L1	8K	4	32	
L1	1K	2	16	
L2	1024K	16	128	
L2	256K	16	64	
L2	128K	16	64	

DRAM	512MB	1	16	

Step 4: Use perf tool to measure energy

SKX machine on Stampede2 supports three perf events (power/energy-cores/, power/energy-pkg/, power/energy-ram/) to measure energy. Intel Xeon processors now only support "package" and "dram" domains only. The "cores" domain is no longer supported, which means you can only measure energy consumed in the whole compute node. Use "-a" option with perf events (ex. `perf -a -e power/energy-cores/, power/energy-pkg/, power/energy-ram/ -- ./sleep 10`) to measure energy. You will see the amount of energy consumed while you do not actually run anything. We will measure the energy to compute matrix multiplication by subtracting the energy while idling from the energy while running matrix multiplication.

Question 11

Now combine the access count (table in Q7) and per-access energy (table in Q10) together to derive energy consumption for matrix multiplication based on our model. Fill in both measured and model results in the cache-aware and cache-oblivious columns as fraction of original. Also, compare the model results with the measurement (with the average number of multiple measurements). Are the measured numbers reasonable? What conclusions can you draw?

N=M=P	L1 Size	L1 Tile	L2 Size	L2 Tile	Original Energy		Cache Aware / Original		Cache Oblivious / Original	
					Model	Measured	Model	Measured	Model	Measured
	Your processor params									
4096	32	64	1024	128						
512	8	32	256	64						
512	1	16	128	64						

Question 12

What is the reduction in power consumption among original, cache-aware and cache oblivious (think about this carefully)?