BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJASTHAN)

CS F422 – Parallel Computing
Lab#1

Note: Please use programs under *Code* directory supplied with this sheet. Do not copy from this sheet.

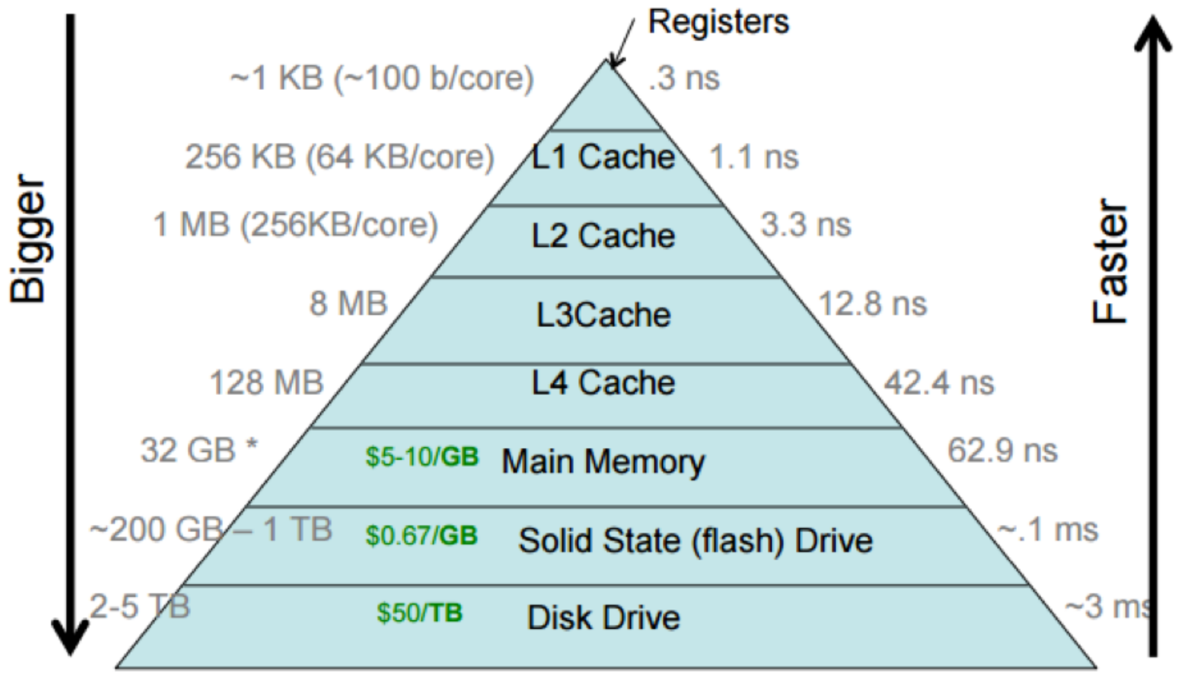
The lab has the following objectives:

1. Understand why caching matters
2. Understand and experiment with cache access patterns
3. Caching impact on Matrix multiplication example

Caching

Computers often use a technique called caching to make a memory system comprised of a hierarchy of storage devices to appear as large and fast. In particular, when we build a cache, we use a small amount of relatively fast storage at one level of the memory hierarchy to speed up access to a large and relatively slow storage at the next lower level of the memory hierarchy. By "lower", we mean further from the processor.

Here's an example picture of what the memory hierarchy in your computer might look like:



At the top of the memory hierarchy, the picture shows the processor cache divided into multiple levels, with the L1 cache (sometimes pronounced “level-one cache”) closer to the processor than the L4 cache. This reflects how processor caches actually work in practice (there really are 3-4 different caches in your processor!), but we often think of a processor cache as a single unit.

Different computers have different sizes and access costs for these hierarchy levels; the ones listed above are typical. For example, a common desktop computer with four cores (i.e., four independent processors) might have ~200 bytes of registers; ~9 MB of processor cache; 8 GB primary memory; 512 GB SSD; and 2 TB hard disk. The processor cache divides into three levels: e.g., there might be 128KB of L1 cache, divided into four 32KB components; 512KB of L2 cache, divided into two 256KB components (one per “socket”, or pair of cores); and 8MB of L3 cache shared by all cores.

Each layer in the storage hierarchy acts as a cache for the following layer.

How is caching used?
Caches are so critical to computer systems that it sometimes seems like caching is the only performance-improving idea in systems. Processors have caches for primary memory. The operating system uses most of primary memory as a cache for disks and other stable storage devices. Running programs reserve some of their private memory to cache the operating system’s cache of the disk.

People have made entire careers out of proposing different variants of caches for different use cases. When a program processes a file, the actual computation is done using registers, which are caching values from the processor cache, which is caching data from the running program’s memory, which is caching data from the operating system, which is caching data from the disk. And modern disks contain caches inside their hardware too!

Why Caching Matters?

We’ll use simple I/O benchmark programs to evaluate the real-world impact of caches. Specifically, we’ll run several programs that write data to disk (to your computer’s hard drive). Each of these programs accomplish the same task of writing 5,120,000 bytes (5 MB) to a file called data, but they do so in several different ways.

Synchronous writes one byte at a time
Take a look at `w01-byte.c`. Currently, `w01-byte.c` writes to a file called `data`:

- one byte at a time
- using direct system calls (`write`)
- synchronously to the disk

"Synchronously" means that each write operation completes only when the data has made its way through all layers of caching mentioned above out to the true disk hardware. We request synchronous writes by opening the file with the `O_SYNC` option to `open` (line 5). You can compile and run `w01-byte.c` by running:

- `make w01-byte` to compile
- `./w01-byte` to run

Q?

1. Run `w01-byte` and record how many bytes per second it can write. You'll notice that this is quite slow.

<u>Asynchronous writes one byte at a time</u>

We can optimize the previous program by removing the O_SYNC flag, so that the program requests "asynchronous" writes, and relies on the operating system to write the data to the disk when it's convenient (but after your write call has returned).

Q?

1. Copy and paste the contents of w01-byte.c into a new file called w02-byte.c, and add w02-byte.c to the EXEC := ... line in the Makefile.

   Remove the O_SYNC flag from the open call in w02-byte.c.

   Once again, compile (using make w02-byte), run the program, and record its write speed in bytes per second to your answers.md.
2. How does it compare to the speed of w01-byte.c
3. Can you explain the difference?

<u>Adding more caching layers</u>

While this new asynchronous program is much faster, it's still being slowed down by expensive operations – namely system calls. These operating system invocations are slow, because they require the processor to stop what it's doing, save the user program's state, give control to the operating system to perform a write or read call, and then switch back.

To get around this, we can use an additional layer of caching through library calls – function calls that are implemented in the standard library and keep their own cache. These functions avoid the cost of switching states to the operating system, because you're writing to their in-memory cache rather than to the OS's. By using the standard library call *fwrite* rather than the write system call, the program performs writes to a cache in program memory without involving the operating system.

Q?

1. Create a copy of w01-byte.c and rename the file w03-byte.c.

   Modify the new file to use the fopen, fwrite, and fclose library calls (instead of open, write, and close) and then record the results. (Make sure to add w03-byte to the Makefile.)

   Note that this may involve changing the return values and arguments of the open, write, and close calls.

2. How does it compare to the speed of w02-byte.c
3. Can you explain the difference?

How the Processor Cache Works?

Now that you have a better idea why caching is so important, let's get into how one particular, important kind of cache in your computer works. We will be looking at the processor cache, which is a fast piece of memory that the processor uses to be able to access data in memory more quickly.

Note that the idea behind caching is general and applies both to the I/O caching you explored in previous section and to the caching we will explore in this exercise. We will use a cache simulator tool, called Venus, to visualize changes to the cache.

Caches have several hit, miss, and eviction policies, each with their own trade-offs. In the Venus simulator, you will use a write-through hit policy, a write-allocate miss policy, and an LRU eviction policy. We explain what these terms mean below!

- Write-through means that on a write "hit", data is written to both the cache and main memory. Writing to the cache is fast, but writing to main memory is slow; this makes write latency in write-through caches slower than in a write-back cache (where the data is only written to main memory later, when the cache block is evicted). However, write-through caches mean simpler hardware, since we can assume in write-through caches that memory always has the most up-to-date data

- Write-allocate means that on a write miss (where data you're writing to isn't already in the cache), you pull the block you missed on into the cache and perform the write on it.
- LRU (Least recently used) means that when a cache block must be evicted to make space for a new one, we select the block that has been used furthest back in time ("least recently") and throw it out of the cache.

This lab uses the cache.s file. This file contains assembly code in RISC-V assembly, a an assembly language. You won't need to understand how to read this assembly code, but if you're curious about the differences between the x86-64 and RISC-V architectures, check out this post. Here is some C-like pseudocode for the assembly instructions in cache.s:

```
int array[];  // Assume sizeof(int) == 4
for (k = 0; k < repcount; k++) { // repeat the loop repcount times
    // Step through the selected array segment with the given step size.
    for (index = 0; index < arraysize; index += stepsize) {
        if(option==0)
            // Option 0: One cache access - write
            array[index] = 0;
        else
            // Option 1: Two cache accesses - read AND write
            array[index] = array[index] + 1;
    }
}
```

Lines 24-28 of cache.s allow us to set the arraysize, stepsize, repcount, and option variables. *li* is a RISC-V instruction to load a constant into a register.
Currently:

- the arraysize, represented by a0 in the assembly code stores 256 bytes, which is 64 integers (because sizeof(int) = 4 in the Venus simulator).
- the stepsize represented by a1 stores 2
- the repcount represented by a2 stores 1
- the option represented by a3 stores 1

Make sure you understand what the pseudocode does and how you can edit the variables before you proceed to analyze cache configurations on it.

- The most important thing to understand is the pseudocode. When you run cache.s, instructions that perform this pseudocode will be executed.
- Which elements you access is determined by the step size (a1) and how many times you do so is determined by the repcount (a2). These two parameters will most directly affect how many cache hits or misses will occur. The option (a3) will affect whether you zero out some elements of some array (option 0) or increment them (option 1).

Q?

1. Open the Venus Cache Simulator.

   Copy and Paste the code from cache.s into the Editor tab.

   In the Simulator tab, click Assemble and Simulate from Editor to assemble the code.

   Once you've assembled the code, you can click Run to execute the code. You can also click on assembly instructions to set breakpoints in the code. Once you've run the program, you need to click Reset before you can run it again.

2. You can find the cache hit rates in the "Cache" sidebar on the right hand side of the Simulator tab (you may have to scroll down a bit to see the "Hit Rate" row).
3. Try making changes in cache.s and cache configuration as per the table and observe cache hits/misses.

Scenario No	Program parameters	Cache Parameters	Remark
1	• Array Size (`a0`): 128 (bytes) • Step Size (`a1`): 1 • Rep Count (`a2`): 2 • Option (`a3`): 0	• Cache Levels: 1 • Block Size: 8 • Number of Blocks: 1 • Enable?: Click on the button to make it green • Placement Policy: Direct Mapped • Associativity: 1 • Block Replacement Policy: LRU	Set a breakpoint at the instructions 0x30 and 0x40 in the Simulator by clicking on them. Instruction 0x40 corresponds to the instruction that performs array[index] = 0; in option 0, and 0x30 corresponds to the instruction that performs array[index] = array[index] + 1; under option 1.

Q?

1. How many array elements can fit into a cache block?
2. What combination of parameters is producing (i.e., explains) the hit rate you observe? Think about the sizes of each of the parameters.
3. What is our hit rate if we increase Rep Count arbitrarily? Why?

Scenario No	Program parameters	Cache Parameters	Remark
2	• Array Size (a0): 128 (bytes) • Step Size (a1): 27 (step by 27 elements of the array) • Rep Count (a2): 2 • Option (a3): 0	• Cache Levels: 1 • Block Size: 8 • Number of Blocks: 1 • Enable?: Click on the button to make it green • Placement Policy: Direct Mapped • Associativity: 1 • Block Replacement Policy: LRU	Set a breakpoint at the instructions 0x30 and 0x40 in the Simulator by clicking on them. Instruction 0x40 corresponds to the instruction that performs array[index] = 0; in option 0, and 0x30 corresponds to the instruction that performs array[index] = array[index] + 1; under option 1.

Q?

1. What combination of parameters is producing (i.e., explains) the hit rate you observe? Think about the sizes of each of the parameters.
2. What happens to our hit rate if we increase the number of blocks and why?

Scenario No	Program parameters	Cache Parameters	Remark
2	• Array Size (a0): 256 (bytes) • Step Size (a1): 2 • Rep Count (a2): 2 • Option (a3): 1	• Cache Levels: 1 • Block Size: ? • Number of Blocks: ? • Enable?: Click on the button to make it green • Placement Policy: Direct Mapped • Associativity: 1 • Block Replacement Policy: LRU	Venus will throw an error if your number of blocks and block size is not a power of 2

Q?

1. Choose a `number of blocks` greater than `1` and determine the smallest `block size` that uses every block *and* maximizes the hit rate given the parameters above. Explain why.

Writing Cache Friendly Code

Hopefully, you realized in the last exercise that we can maximize our hit-rate by making the cache large enough to hold all of the contents of the array we're accessing. In practice, it would be nice if we could store all of the contents of the memory that we're accessing in the fastest possible storage, but like many things in life, we're often limited by financial resources (remember the memory hierarchy). You can check how much space in bytes your machine is allocating for caching by typing getconf -a | grep CACHE or lscpu.

By understanding how caches work, you can optimize a program's memory access patterns to obtain good performance from the memory hierarchy. As a concrete example of such optimization, let's analyze the performance of various matrix multiplication functions. Matrix multiplication operations are at the heart of many linear algebra algorithms, and efficient matrix multiplication is critical for many applications within the applied sciences and deep learning fields (for example, much of what TensorFlow gets up to during model training and inference under the hood turns into matrix multiplications).

If you recall, matrices are 2-dimensional data structures wherein each data element is accessed via two indices. To multiply two matrices, we can simply use 3 nested loops, assuming that matrices `A`, `B`, and `C` are all n-by-n and stored in one-dimensional column-major arrays (see below for an explanation of what these terms mean):

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            // equivalent to C[j][i] += A[k][i] * B[j][k]
            // if these were 2D arrays
            C[i+j*n] += A[i+k*n] * B[k+j*n];
```

Note: In this exercise, the 2D matrix is flattened to a one-dimensional array. For example, if you had a 5x5 matrix, it would be a 25 element 1D array, where each row is consecutively laid out in memory. If you generally access the 2nd row and 3rd column of the 5x5 matrix called A using A[1][2] (since it's 0 indexed), you'd access the equivalent entry of the 1D array called B with B[1*5 + 2].

In the above code, note that the loops are ordered `i`, `j`, `k`. If we examine the innermost loop (the one that increments `k`), we see that it…

- moves through B in row-order (each iteration of the innermost loop iterates through a single row of matrix B)
- moves through A in column-order (each iteration of the innermost loop iterates through a single column of matrix A)
- accesses one entry of C

Checkout matrixMultiply.c. You'll notice that the file contains multiple implementations of matrix multiply with three nested loops. You can compile and run matrixMultiply.c by running:

- `$ make matrixMultiply`
- `$ ./matrixMultiply`

While each of these implementations will correctly calcuate the matrix multiplication, the order in which we choose to access the elements of the matrices can have a large impact on performance. From the previous exercises, you should realize that caches perform better (more cache hits, fewer cache misses) when memory accesses take advantage of temporal locality (recently accessed blocks are accessed again) and spatial locality (nearby blocks are accessed soon), utilizing blocks already contained within our cache.

Q?

Run the program a couple of times, order the functions from fastest to slowest, and explain why each function's ranking makes sense using your understanding of how the cache works. Some functions might have similar runtimes. If this is the case, explain why.

Note: The unit "Gflops/s" reads, "Giga-floating-point-operations per second." The bigger the number, the faster the program is running.

Acknowledgements:
This lab sheet is adapted from the labsheets given with the book "Computer System from Programmers Perspective".

End of lab1