CS209P Project Phase 3

April 3, 2025

- This will be the final phase of the project, and is involved, so start early. Deadline: May 12th. Keep yourself free for a zoom call or in-person (if you are still in campus) on May 13th for the evaluation. Please note that there will be no extensions as I will have to submit the grades.
- Extend the simulator that you developed in Phase 2 to incorporate Cache.
- A memory access will now first search for the address in the cache. On a miss, the data will be fetched from the main memory.
- This effectively means that the memory instructions (loads and stores) will not be completed in one cycle.
- Loads and Stores will have variable latency, and hence the penalty (stalls) due to the memory access is variable.
- The simulator should be able to simulate a cache, with two different cache replacement policies. One is LRU. You can choose the other policy.
- The input along with the assembly code will include cache size, block size, associativity, and access latency of the cache. These parameters can be provided in a separate file as input.
- The input will also include main memory access time.
- At the end of the execution, the simulator should output the number of stalls, cache miss rate, and the IPC (Instructions per Cycle).
- Cache design:
  - Two levels of cache.
  - First level: two caches, one for instructions and one for data. We will refer to them as L1I and L1D.
  - Second level: a single unified cache. We will refer to it as L2.
  - An instruction fetch will also be considered as a memory access. If hit in the cache, the instruction will be fetched from the cache. If miss, it will be fetched from the main memory. Hence the IF stage will also have variable latency. Note that a cache block of 64 bytes can hold 16 instructions.

Recall from the previous phase:

- You merged the 4 compute units How? There will be only one fetch unit for all the compute units. But each compute unit will have its own decode/register fetch, execute, memory and writeback stages. The compute units will share the instruction memory and data memory. How does this work? Remember that each compute unit had a special purpose register, let us refer to it CID. Assume the following code snippet:

```
1     ADD x2, x3, x4 # all compute units will execute this
          instruction
2     BNE cid, 1, Label # all compute units will execute this
          instruction, but only the compute unit with CID=1 will
          take the branch
3     ADD x5, x6, x7 # all compute units will fetch this
          instruction because there is only one fetch unit. But CID
          =1 will not execute this instruction. The instruction is
          simply ignored after the fetch stage in CID=1. The other
          compute units will execute this instruction.
4   Label: ADD x8, x9, x10 # all compute units will execute this
       instruction
```

- One of the test cases for this phase will be array addition. The below example is given in C code, but a direct RISC-V assembly code can be used as a test case.

```
1   for (i = 0; i < 100; i++) {
2     sum += a[i];
3   }
4   print(sum);
```

  All the four compute units should be able to execute this code. But compute unit 1 will compute the sum of the first 25 elements, compute unit 2 will compute the sum of the next 25 elements, and so on. The final sum should be the sum of all the elements in the array. Only compute unit 1 will print the sum. You can convert the above C code to RISC-V assembly code as you see fit.

- If you are unable to complete the project, do prepare a document detailing what you tried, and what did not work etc. The document along with the incomplete code will be evaluated.

- Well, we noted that the compute units will share the instruction memory and data memory. And in the C code each one is updating a variable sum. So, how do we ensure that the sum is updated correctly? Because the sum is a shared variable, we need to make sure that when one compute unit is updating it, the other compute units are not interfering. This can be achieved by having each compute unit maintain its own partial sum in a separate memory, and then combining these partial sums at the end. So the code will look like:

```
1  for (i = 0; i < 100; i++) {
2    sum[CID] += a[i]; // each compute unit maintains its own
        partial sum
3  }
4  // Tag1
5  if (CID == 1) {
6    for (i = 2; i <= 4; i++) {
7      sum[1] += sum[i]; // compute unit 1 combines the partial sums
```

```
8     }
9     print(sum[1]); // only compute unit 1 prints the sum
10 }
```

- Would this work?
- Look at the line which says Tag1. What is the guarantee that when CID=1 is executing the code after Tag1, the other compute units are still updating their partial sums?
- How do you ensure that the other compute units have completed updating their partial sums before compute unit 1 starts combining them?
- So your hardware should support a special instruction. Let's call it `SYNC`. When a compute unit executes this instruction, it should wait until all the other compute units have also executed this instruction. Only then should it proceed.
- The `SYNC` instruction should be a no-op. That is, it should not do anything other than waiting for the other compute units to execute it.
- How do you implement this? Your report should have a detailed explanation of how you implemented this and what are the design choices you made. Note, you cannot just implement a simulator hack; your design should be such that it can be implemented in the context of the hardware you are simulating. You cannot also just wait for a fixed number of cycles.

- In addition to the L1 instruction (L1I) and data (L1D) caches at the first level of the memory hierarchy, introduce an additional fast memory unit, referred to as `scratchpad memory`. This memory has the same access latency and size as the L1D cache. However, unlike the cache, the scratchpad memory is entirely programmer-controlled: the programmer is responsible for determining which data to store and which data to evict. No automatic search or replacement mechanism is performed in this memory.
- To utilize the scratchpad memory (SPM), you can use two special instructions: `lw_spm` and `sw_spm`. These instructions allow direct load and store operations to SPM.
- `lw_spm rd, offset(rs1)`: This instruction reads a word from scratchpad memory at the computed address and stores it into register `rd`.
- Operation: `rd` ← `SPM[rs1 + offset]`.
  - `rd`: Destination register where the loaded value will be stored.
  - `rs1`: Base address register that holds the starting address in SPM.
  - `offset`: Immediate value added to `rs1` to compute the effective address which should not exceed scratchpad memory capacity.
- `sw_spm rs2, offset(rs1)` : This instruction writes the value from register `rs2` into scratchpad memory at the computed address.

- Operation: `SPM[rs1 + offset]` ← `rs2`.
  - `rs2`: Source register containing the value to be stored.
  - `rs1`: Base address register that holds the starting address in SPM.
  - `offset`: Immediate value added to `rs1` to compute the effective address which should not exceed scratchpad memory capacity.

Listing 1: Strided array addition without using scratchpad memory

```
1  // assume that the array stores word-sized data elements
2  int X = L1D_cache_size_in_bytes / size_of(a[i]); // i.e. X = L1D_
       cache_size_in_bytes / number_of_bytes_in_a_word (4 bytes in
       your case)
3  // assume that the sum[] is stored in registers
4
5  for(count = 0; count < 100; count++) {
6    for (i = 0; i < 100; i++) {
7      //assume array out of index never occurs
8      //goal is to find the sum of a[0], a[X], a[2*X], ..., a[98*X
           ], a[99*X]
9      sum[CID] += a[i * X]; // each compute unit maintains its own
           partial sum
10   }
11 }
12
13 // Tag1
14 if (CID == 1) {
15   for (i = 2; i <= 4; i++) {
16     sum[1] += sum[i]; // compute unit 1 combines the partial sums
17   }
18   print(sum[1]); // only compute unit 1 prints the sum
19 }
```

Listing 2: Strided array addition using scratchpad memory

```
1  // assume that the array stores word-sized data elements
2  int X = L1D_cache_size_in_bytes / size_of(a[i]); // i.e. X = L1D_
       cache_size_in_bytes / number_of_bytes_in_a_word (4 bytes in
       your case)
3  // assume that the sum[] is stored in registers
4
5  //assume SPM[] is word addressable here
6  //fill scratch pad memory
7  for (i = 0; i < 100; i++) {
8   //assume array out of index never occurs
9   //goal is to find the sum of a[0], a[X], a[2*X], ..., a[98*X], a
        [99*X]
```

```
10   SPM[i] = a[i * X]; //use sw_spm instruction to store in
         scratchpad memory
11 }
12
13 for(count = 0; count < 100; count++) {
14   for (i = 0; i < 100; i++) {
15     sum[CID] += SPM[i]; // use lw_spm instruction
16   }
17 }
18
19 // Tag1
20 if (CID == 1) {
21   for (i = 2; i <= 4; i++) {
22     sum[1] += sum[i]; // compute unit 1 combines the partial sums
23   }
24   print(sum[1]); // only compute unit 1 prints the sum
25 }
```

- Part 1:
  - Assume a direct mapped L1D cache of size 400 bytes and the equivalent SPM to also be 400 bytes.
  - For the above test case of strided array addition in 1, 2, compare the number of clock cycles required to execute the assembly implementation of the following two algorithms:
    1. Algorithm 1: Implementation without using scratchpad memory.
    2. Algorithm 2: Implementation with the utilization of scratchpad memory.
- Part 2:
  - Assume a fully associative L1D cache of size 400 bytes and the equivalent SPM to also be 400 bytes.
  - Repeat the steps in Part 1.
  - Does it look like there is no performance difference?
- Part 3:
  - If there was no performance difference in Part 2, it appears to be the case that SPM and a fully associative structures of same size have similar performance.
  - No, generally the latencies (and power requirements) of SPMs are far lower than caches. Write a paragraph in your submission as to why?
  - Ignoring the latency (and power) i.e., assuming the same latency for SPM and L1D cache, is there a performance benefit if I use SPM?
  - HINT: Instead of accessing 100 elements in the array, access 200 elements i.e., change the for loop in line 6 of 1 and line 14 of 2 to 200, and run the programs 2 and 1. You will have to modify 2, such that only 100 elements are stored in SPM (when you fill it).