Assignment 2 - Introduction to High Performance Computing Systems

Felipe Gómez Lozada

October 14, 2024

1

a

For perfect operand pre-fetching the performance bottleneck is given only by the memory bandwidth, since by sending the information to the processor beforehand we remove the delay of the memory latency. Given that we also have unlimited multithreading, we can assume that one thread executes the operation while another receives the next operand, and then peak performance occurs when the processor can receive an element (4 bytes) for the sum on each clock cycle (2 GHz). This corresponds to a peak performance of 2 GFLOPS and a memory bandwidth of at least

$$4\,\text{byte} \times 2\,\text{GHz} = 8\,\frac{\text{GB}}{\text{s}}. \tag{1}$$

b

To calculate the maximum performance in this case consider the following sequence: First, the processor requests a cache line (4 words), since we assume infinite memory bandwidth the delay corresponds only to the memory latency which corresponds to 80 processor clock cycles. Then if we neglect the communication time between the cache and the processor on the following four clock cycles it will add the complete cache. Therefore, for each 84 clock cycles, the processor can perform 4 FLOP, corresponding to a maximum performance of

$$\frac{4\,\text{FLOP}}{84\,\text{clock cycles}} \times \frac{2\,\text{clock cycles}}{1\,\text{ns}} = 95.24\,\text{MFLOPS}, \tag{2}$$

which corresponds to a fraction of 0.048 of the peak performance.

2

Considering the following system characteristics:

- CORE 0: Threads 0 and 1.
- CORE 1: Threads 2 and 3.
- CORE 2: Thread 4.
- CORE 3: Thread 5.
- Cacheline FOO: Variables A and B.
- Cacheline BAR: Variables C and D.

The simulation of the coherent cache lines using the MSI protocol corresponds to:

OPERATION	Cacheline	CORE 0	CORE 1	CORE 2	CORE 3
Initial	FOO	I	I	I	I
	BAR	I	I	I	I
Thread 0:	FOO	S	I	I	I
Load A	BAR	I	I	I	I
Thread 1:	FOO	S	I	I	I
Load B	BAR	I	I	I	I
Thread 2:	FOO	S	I	I	I
Store C	BAR	I	M	I	I
Thread 0:	FOO	M	I	I	I
Store A	BAR	I	M	I	I
Thread 3:	FOO	M	I	I	I
Load C	BAR	I	M	I	I
Thread 3:	FOO	I	M	I	I
Store A	BAR	I	M	I	I
Thread 4:	FOO	I	S	S	I
Load B	BAR	I	M	I	I
Thread 5:	FOO	I	S	S	I
Load D	BAR	I	S	I	S
Thread 0:	FOO	S	S	S	I
Load B	BAR	I	S	I	S

3

On one hand, cache coherence corresponds to synchronizing cache memory among different processors. This is an OS task, since cache memory is not directly controllable by the programmer, and multiple protocols which enforce it have the same result overall, to avoid different stored values for the same variable in different caches. On the other hand, memory consistency corresponds to the unambiguous storing of values along a program which could be affected by race conditions among processors that result in multiple outputs for a single program. In this case, the OS has different paradigms for which memory consistency is enforced to a certain level, leaving exceptions that have to be taken into account by the programmer. For example, most processes in an SC system are memory-consistent (exceptions are race conditions outside the control of the OS) since it enforces a specific order of operations. In contrast, for a TSO or RC, some operations can be reordered to improve the program's efficiency, which could lead to the memory being inconsistent if the programmer is not careful.

Next, we provide some examples of each:

- Consider the following program for a cache-coherent, TSO system, where we assume that `x` and `y` are mapped to different cache lines:

```
x = 0;
y = 0;
if (pid = 0){
    x = 1;
    print y;
}
if (pid = 1){
    y = 1;
    print x;
}
```

  According to the pseudo-code we do not expect the program to result in `x=0;y=0`, while it can return other possible results due to the race condition such as `x=0;y=1`, `x=1;y=0` or `x=1;y=1`. Nevertheless, for a TSO system each processor can send the corresponding store instruction to the store buffer, and while this is processed the load of the other processor reads the value of the variable in main memory, for which the storing has not reached yet, resulting in `x=0;y=0`. If on the contrary, we assume that the system is not cache coherent and SC, given that both variables are mapped to different cache lines there is a cache miss on each load and store within each

processor, meaning that all the information is synchronized by the main memory and program works as expected.

- Now consider the following code for a SC system without cache coherence where both `x` and `y` are mapped to the same cache line:

```
x = 0;
y = 0;
if (pid = 0){
    print x;
    x = 1;
}
if (pid = 1){
    print y;
    y = 1;
}
Barrier, wait for all processes;
print x+y;
```

Here the expected result is `0,0,2,2`. Even then, when each core prints the corresponding variable they store it on the cache and then update it locally, meaning that after the barrier, since the system is not cache-coherent, there will be a cache read hit for both `x,y` in each core that will use the original value of the variable which was not updated, resulting in `0,0,1,1` after execution. On the other hand, if there was cache coherence but we raised the consistency of the memory to a RC system, the barrier would synchronize operations on different addresses and the result would be as expected.

4

In the first case, the objective is to reuse the filter in the local calculation of the convolution, then if we have two rows belonging to different fmaps we can summarize the application of the filter to both by concatenating both rows and applying a convolution between this with the filter, resulting in a larger row belonging to two different concatenated psums. On the other hand, if we want to reuse the fmap, by creating a larger filter, the product of two different interleaved filters, the resulting convolution will generate an also interleaved row between the results for the different psums. In both situations, the properties of the convolution allow a simplification of multiple operations in a single one.