Homework 2

School of Data and Computer Science

17341015 Hongzheng Chen

I think this homework exists lots of uncertainty. We do not know what techniques in *modern CPU* should be considered. Like forwarding and superscalar, though the problem is not explictly mentioned, but they are both very important techniques in *modern* CPU. Moreover, only providing the machine configuration is not enough, the compiler can also affect the performance. If some compiler techniques like loop unrolling or software pipelining are used, the results will be different. Thus, I can only provide the answer in my understanding.

1.Instruction-Level Parallelism

Suppose you have a machine M_1 with two load/store units that can each load or store a single value on each clock cycle, and one arithmetic unit that can perform one arithmetic operation (e.g., multiplication or addition) on each clock cycle.

A. Assume that the load/store and arithmetic units have latencies of one cycle. How many clock cycles would be required to execute `computeInnerProduct` as a function of N? Explain what limits the performance.

Answer. We label the instructions as following.

```
inline void innerProduct (point *a, point *b, float *result)
{
    float x1 = a->x; // 1, load from memory
    float x2 = b->x; // 2
    float product1 = x1*x2; // 3, store in register
    float y1 = a->y; // 4
    float y2 = b->y; // 5
    float product2 = y1*y2; // 6
    float inner = product1 + product2; // 7
    *result = inner; // 8
}
```

If no parallelism is considered, 4 loads, 2 multiplications, 1 addition and 1 store need 8 cycles per loop. Thus, $8N$ cycles for `computeInnerProduct`.

One cycle issuing one instruction limits the performance.

B. Now assume that the load/store and arithmetic unit have latencies of 10 clock cycles, but they are fully pipelined, able to initiate new operations every clock cycle. How many clock cycles would be required to execute `computeInnerProduct` as a function of N? Explain how this relates to your answer to part A.

Answer. Assume no forwarding is considered. Since there are two load/store units, the 2nd instruction can be issued in the 2nd cycle. But the third one should be wait for the first two instructions due to the dependency relation. Then, 1+10+10=21 cycles are needed for the first three.

Although out-of-order execution is allowed, the limitation of load/store units does not allow the 4th and 5th instruction to be executed simultaneously. Therefore, 21 cycles are also needed for 4→6 instructions.

Since the load/store units are free after the first 10 cycles, the first load/store can be used in the 11th cycle, and the second one can be used in the 12th cycle. That is to say, the 4th and 5th instruction can be hided behind the execution of 1 $\sim$ 3 instructions (See Fig. 1).

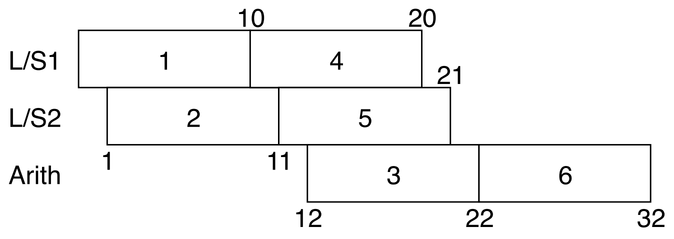


Figure 1: Execution graph of Part B

In total, (21+1+10+10+10)N=52N cycles are required to execute `computeInnerProduct`.

Part B is able to initiate new operations every clock cycle, thus the performance imporves. The new limitation is the number of I/O ports.

2.SIMD with ISPC

Suppose machine M_2 has one 8-wide SIMD load/store unit, and one 8-wide SIMD arithmetic unit. Both have latencies of one clock cycle.

A. How many clock cycles would be required to execute `computeInnerProductISPC` as a function of N? Explain what limits the performance.

Answer.

  - Loading `A[i].x`, `B[i].x`, `A[i].y`, `B[i].y` needs one cycle.
  - Adding two terms needs one cycle.
  - Storing the result into `result[i]` needs one cycle.

  Thus, each loop needs 3 cycles to execute. Notice the loop is `foreach` loop. If the machine has enough CPU cores and compiler fully parallelizes this loop, ignoring the threads creating overheads, then only 3 cycles are needed for the whole loop, since no dependency relation among different loop index.

  If the compiler just views the `foreach` loop as `for` loop, then $3N$ cycles are needed.

  For other cases, I cannot figure out the results due to the lack of information.

B. If we were to run the `computeInnerProductISPC` on a five-core machine M_3, where each core has the same SIMD capabilites as M_2, what would be the best speedup it could achieve over the single-core performance of part A? Explain.

   Answer. We can divide the task into 5 subtasks, say $0 \sim N/5$ for core 1, $N/5 \sim 2/5N$ for core 2, …, $4/5N \sim N$ for core 5. Since the SIMD capability in each core is the same, the best speedup would be 5x.

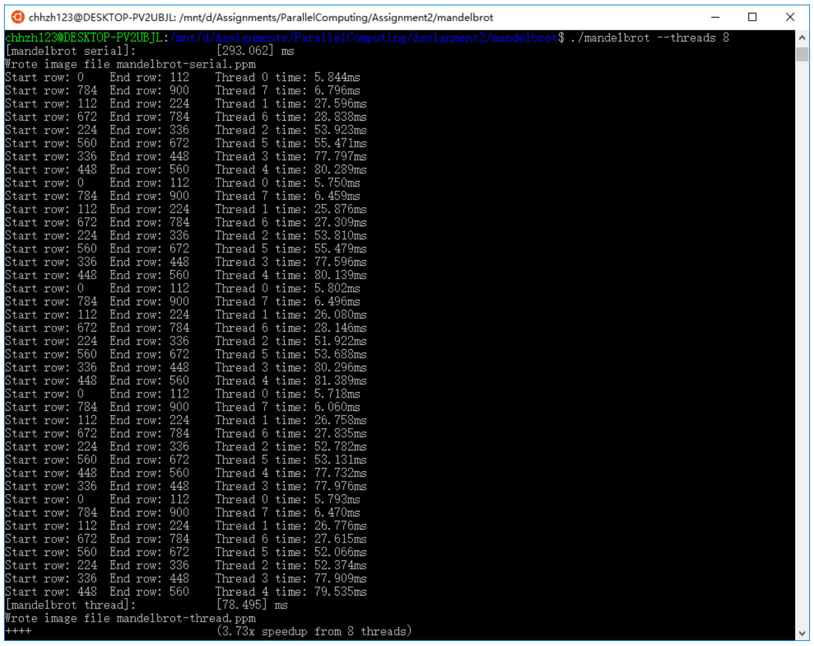
3.Parallel Fractal Generation Using Pthreads

Experimental setting:

- Ubuntu 18.04(LTS) + gcc 7.3.0
- Intel Core i7-7700HQ 2.80GHz (8 cores)
- 8GB memory

For detailed implementation, please refer the codes in `mandelbrot.cpp`.

Due to space limitation, only the output of 8 threads is shown in Fig. 2. Other results can be accessed by building the project.


chhzh123@DESKTOP-PV2UBJL: /mnt/d/Assignments/ParallelComputing/Assignment2/mandelbrot

```
chhzh123@DESKTOP-PV2UBJL:/mnt/d/Assignments/ParallelComputing/Assignment2/mandelbrot$ ./mandelbrot --threads 8
[mandelbrot serial]:            [293.062] ms
Wrote image file mandelbrot-serial.ppm
Start row: 0    End row: 112    Thread 0 time: 5.844ms
Start row: 784  End row: 900    Thread 7 time: 6.796ms
Start row: 112  End row: 224    Thread 1 time: 27.596ms
Start row: 672  End row: 784    Thread 6 time: 28.838ms
Start row: 224  End row: 336    Thread 2 time: 53.923ms
Start row: 560  End row: 672    Thread 5 time: 55.471ms
Start row: 336  End row: 448    Thread 3 time: 77.797ms
Start row: 448  End row: 560    Thread 4 time: 80.289ms
Start row: 0    End row: 112    Thread 0 time: 5.750ms
Start row: 784  End row: 900    Thread 7 time: 6.459ms
Start row: 112  End row: 224    Thread 1 time: 25.876ms
Start row: 672  End row: 784    Thread 6 time: 27.309ms
Start row: 224  End row: 336    Thread 2 time: 53.810ms
Start row: 560  End row: 672    Thread 5 time: 55.479ms
Start row: 336  End row: 448    Thread 3 time: 77.596ms
Start row: 448  End row: 560    Thread 4 time: 80.139ms
Start row: 0    End row: 112    Thread 0 time: 5.802ms
Start row: 784  End row: 900    Thread 7 time: 6.496ms
Start row: 112  End row: 224    Thread 1 time: 26.080ms
Start row: 672  End row: 784    Thread 6 time: 28.146ms
Start row: 224  End row: 336    Thread 2 time: 51.922ms
Start row: 560  End row: 672    Thread 5 time: 53.688ms
Start row: 336  End row: 448    Thread 3 time: 80.296ms
Start row: 448  End row: 560    Thread 4 time: 81.389ms
Start row: 0    End row: 112    Thread 0 time: 5.718ms
Start row: 784  End row: 900    Thread 7 time: 6.060ms
Start row: 112  End row: 224    Thread 1 time: 26.758ms
Start row: 672  End row: 784    Thread 6 time: 27.835ms
Start row: 224  End row: 336    Thread 2 time: 52.782ms
Start row: 560  End row: 672    Thread 5 time: 53.131ms
Start row: 448  End row: 560    Thread 4 time: 77.732ms
Start row: 336  End row: 448    Thread 3 time: 77.976ms
Start row: 0    End row: 112    Thread 0 time: 5.793ms
Start row: 784  End row: 900    Thread 7 time: 6.470ms
Start row: 112  End row: 224    Thread 1 time: 26.776ms
Start row: 672  End row: 784    Thread 6 time: 27.615ms
Start row: 560  End row: 672    Thread 5 time: 52.066ms
Start row: 224  End row: 336    Thread 2 time: 52.374ms
Start row: 336  End row: 448    Thread 3 time: 77.909ms
Start row: 448  End row: 560    Thread 4 time: 79.535ms
[mandelbrot thread]:            [78.495] ms
Wrote image file mandelbrot-thread.ppm
++++                            (3.73x speedup from 8 threads)
```

Figure 2: Program output of 8 threads

The speedup graph[1] of different number of threads is shown in Fig. 3.

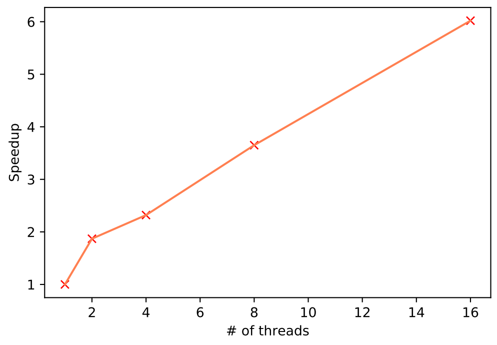


Figure 3: Speedup of different number of threads

From Fig. 2, we can clearly see the load imbalance that some threads terminate fast while others still work hard. Nevertheless, all the threads work parallelly, and the total time is determined by the thread using the most time.

This experiment scales very well. From Fig. 3, we can see that the speedup is almost linear to the number of threads. The more threads, the faster execution. However, the slope of this line is smaller than 1 due to the overheads of creating threads.

[1]This graph is created with Python Matplotlib, and the notebook *plot.ipynb* is also in the attachment.