

Assignment 1

路橙 Lu Cheng 2015010137

Problem 1:

1. See the code in 2.

2. Code is:

```
void* workerThreadStart(void* threadArgs) {
    double startTime = CycleTimer::currentSeconds();
    WorkerArgs* args = static_cast<WorkerArgs*>(threadArgs);
    mandelbrotSerial(args->x0, args->y0, args->x1, args->y1,
        args->width, args->height,
        args->threadId * args->height / args->numThreads, args->height /
args->numThreads,
        args->maxIterations, args->output);
    printf("Hello world from thread %d\n", args->threadId);
    double endTime = CycleTimer::currentSeconds();
    printf("[thread %d ]:\t\t[%.3f] ms\n", args->threadId, (endTime -
startTime) * 1000);
    return NULL;
}
```

In void mandelThread():

```
for (int i = 0; i < numThreads; i++) {
    // TODO: Set thread arguments here.

    args[i].width = width;
    args[i].height = height;
    args[i].maxIterations = maxIterations;
    args[i].numThreads = numThreads;

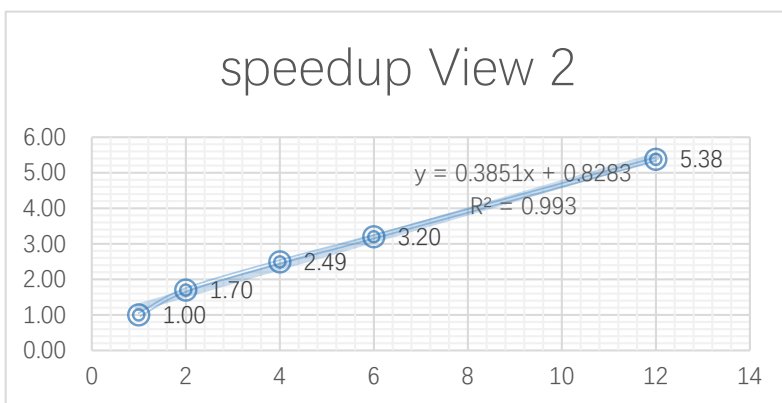
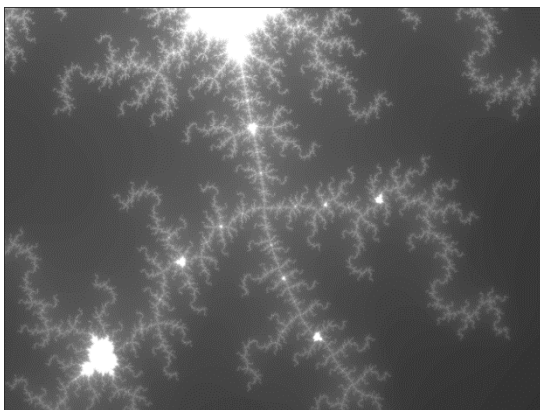
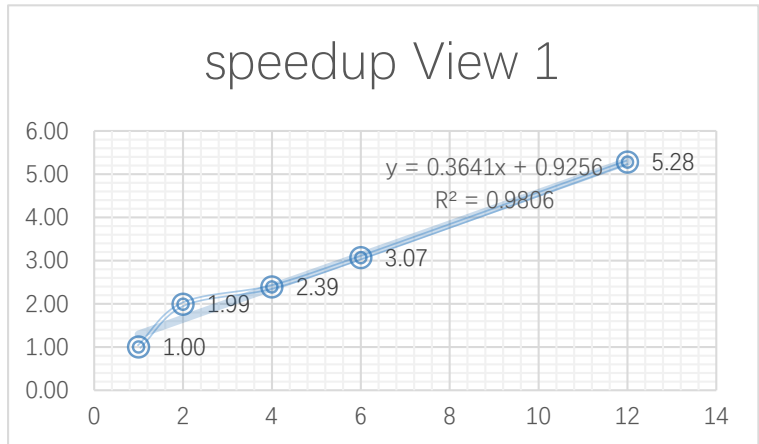
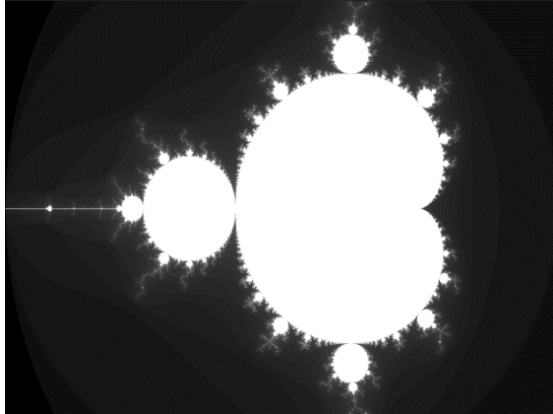
    args[i].x0 = x0;
    args[i].y0 = y0;
    args[i].x1 = x1;
```

```

    args[i].y1 = y1;
    args[i].output = output;

    args[i].threadId = i;
}

```



The speedup of View 1 is not linear in the number of cores. It is because the mid height of this picture needs much more computation than other places. Thus, the work of these threads are not equivalent in amount.

3.

```

[thread 0 ]:      [1.797] ms
[thread 11 ]:      [1.872] ms
[thread 1 ]:      [8.085] ms
[thread 10 ]:      [8.110] ms
[thread 2 ]:      [24.672] ms
[thread 9 ]:      [24.976] ms
[thread 8 ]:      [37.364] ms
[thread 3 ]:      [37.454] ms
[thread 4 ]:      [49.101] ms
[thread 7 ]:      [49.303] ms
[thread 5 ]:      [61.091] ms
[thread 6 ]:      [61.450] ms

```

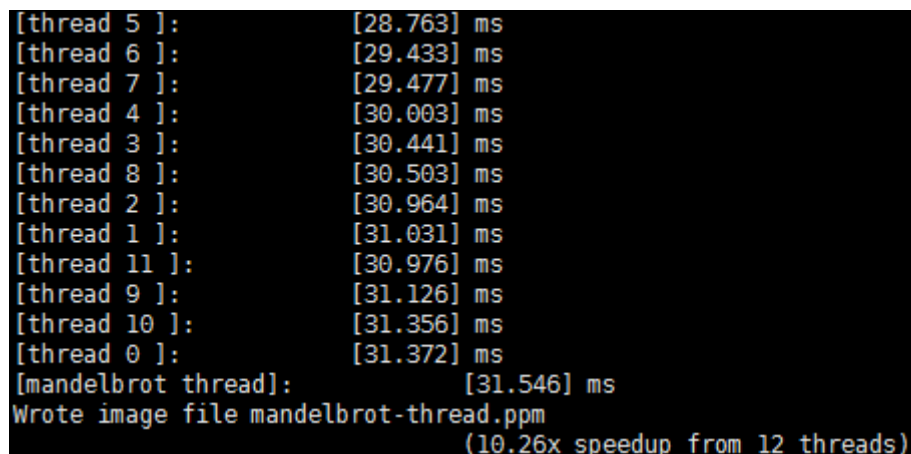
Thread 5 and 6 needs much more work time than thread 0 and 11. So the assignment of work is not even.

4. The code is:

```
void* workerThreadStart(void* threadArgs) {
    double startTime = CycleTimer::currentSeconds();
    WorkerArgs* args = static_cast<WorkerArgs*>(threadArgs);
    float dx = (args->x1 - args->x0) / args->width;
    float dy = (args->y1 - args->y0) / args->height;
    int x0 = args->x0;
    int y0 = args->y0;
    int dd = 10;
    for (int k = dd*args->threadId; k < args->width; k += dd*args->numThreads)
    {
        int startCol = k;
        int endCol = k + dd;
        for (int j = startCol; j < endCol; j++) {
            for (int i = 0; i < args->height; ++i) {
                float x = x0 + j * dx;
                float y = y0 + i * dy;

                int index = (i * args->width + j);
                args->output[index] = mandel(x, y, args->maxIterations);
            }
        }
    }

    printf("Hello world from thread %d\n", args->threadId);
    double endTime = CycleTimer::currentSeconds();
    printf("[thread %d ]:\t\t\t[%.3f] ms\n", args->threadId, (endTime -
startTime) * 1000);
    return NULL;
}
```



```
[thread 5 ]:          [28.763] ms
[thread 6 ]:          [29.433] ms
[thread 7 ]:          [29.477] ms
[thread 4 ]:          [30.003] ms
[thread 3 ]:          [30.441] ms
[thread 8 ]:          [30.503] ms
[thread 2 ]:          [30.964] ms
[thread 1 ]:          [31.031] ms
[thread 11 ]:         [30.976] ms
[thread 9 ]:          [31.126] ms
[thread 10 ]:         [31.356] ms
[thread 0 ]:          [31.372] ms
[mandelbrot thread]:  [31.546] ms
Wrote image file mandelbrot-thread.ppm
(10.26x speedup from 12 threads)
```

Approach: I split every ten column to one thread, and 1,2,...,12 thread compute every ten column one by one. Thus, the computation complexity of every thread is nearly even.

Problem 2:

1. See the code.

2. VECTOR_WIDTH: 2 4 8 16
vector utilization: 77.6% 70.5% 66.7% 65.0%

So the vector utilization decreases. It is because the more width the vector has, the more probability of waiting the other ALU the program has. (Because the total work of every ALU has extremely unequal amount of work).

Problem 3:

Part 1:

1. The maximum speedup I expected is 4x, while the actual speedup is 2.57x.

It is because on one core, the execution unit uses the same fetch/decode unit. When one ALU has finished, it must be waiting for other ALUs until they all have finished their work. As some pixels of the mandelbrot graph needs much more computation than the others, the waiting time leads to the low speedup.

Part 2:

1. The speedup with '-t' is 5.16x, and the speedup that doesn't partition the computation is 2.59x.

2. Finally, the number of tasks is 40. I have chosen 10, 20, 40, 80, 200, and the speedup with the number between 40 and 200 is nearly equal to 25x.

It is because the middle height of the graph needs much more computation

than the other areas that the task with middle area needs too much time. So the ISPC 'thread' which has this task may work in such a long time after other threads have finished. By divide the work into many tasks, every task need little time to do, and all the threads are working together to finish the work.

Problem 4:

1. ISPC speedup is 2.67x, and the speedup with all cores is 27.69x.

2. Very-good-case:

Values[i] = 2.99998f for i=0 to N-1. The speedup is 3.57x for ISPC and 36.82x for task ISPC.

```
[201501019@0001straper prog_sqrt] taskset -C 1,3,5,7,9,11,13,15
[sqrt serial]:      [5115.763] ms
[sqrt ispc]:        [1434.363] ms
[sqrt task ispc]:   [138.933] ms
                  (3.57x speedup from ISPC)
                  (36.82x speedup from task ISPC)
```

The reason is: (1) Setting all the values one value can let SIMD working together and reduce the waiting time. (2) Setting values 2.99998f can increase the computation time so that the time that spawns ISPC threads can take little percentage of total time, so the parallelism speedup can increase.

3. Bad case:

if i % 4 == 0, then values[i] = 2.99998f; else values[i] = 1.0f.

The speedup is 0.92x for ISPC and 9.02x for task ISPC.

```
[sqrt serial]:      [1319.879] ms
[sqrt ispc]:        [1432.774] ms
[sqrt task ispc]:   [146.405] ms
                  (0.92x speedup from ISPC)
                  (9.02x speedup from task ISPC)
```

The reason is:

Every ISPC function generates 4-wide SIMD instructions, so the ISPC 'threads' with value = 1.0f are waiting the computation of value 2.99998f. So the time of every function is the time of computation of value 2.99998f. Since the time of serial sqrt is nearly equal to this time, the speedup is nearly equal to 1x. As the initialization of ISPC function costs a little time, the speedup is less than 1x.

Problem 5

1. The speedup is 3.79x.

```
[saxpy ispc]:      [24.723] ms      [14.465] GB/s   [1.942] GFLOPS  
[saxpy task ispc]: [6.531] ms      [54.762] GB/s   [7.350] GFLOPS  
                  (3.79x speedup from use of tasks)
```

Explain:

Although we split the work into 192 tasks, the limitation of memory bandwidth limits the speed of task ISPC. Also, I think the memory bandwidth is the main limitation, so it is hard to substantially improved. (The data's total memory is 20M.)