Name: Jiaqiang Ruan; ID: jruan

1 Problem 1: Parallel Fractal Generation Using Pthreads

1.1 Modify the code in mandelbrot.cpp

This method achieve 2.00x speedup.

```
void* workerThreadStart(void* threadArgs) {
    WorkerArgs* args = static_cast<WorkerArgs*>(threadArgs);

    if (args->threadId==0)
    mandelbrotSerial(args->x0, args->y0, args->x1, args->y1, args->width, args
        ->height,
    0,args->height/2, args->maxIterations, args->output);
    else if (args->threadId==1)
    mandelbrotSerial(args->x0, args->y0, args->x1, args->y1, args->width, args
        ->height,
    args->height/2, args->height/2+args->height%2,args->maxIterations, args
        ->output);
    return NULL;
}
```

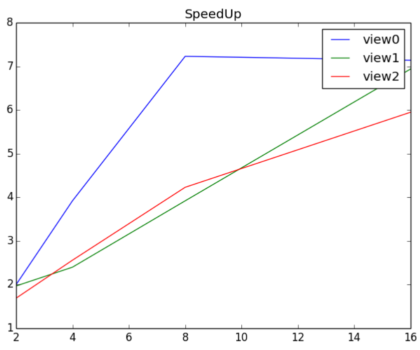
1.2 Extend your code to utilize T threads

```
void* workerThreadStart(void* threadArgs) {

    WorkerArgs* args = static_cast<WorkerArgs*>(threadArgs);
    int interval=args->height/args->numThreads;
    if (args->threadId==args->numThreads-1){
        mandelbrotSerial(args->x0, args->y0, args->x1, args->y1, args->width,
            args->height,
        args->threadId*interval, args->height-args->threadId*interval,
        args->maxIterations, args->output);
    }else{
        mandelbrotSerial(args->x0, args->y0, args->x1, args->y1, args->width,
            args->height,
        args->threadId*interval, interval, args->maxIterations,
        args->output);
    }
    return NULL;
}
```

produce a graph of speedup

See Below.

Is speedup linear in the number of cores used? hypothesize why this is (or is not) the case?



View	Thread	SpeedUp
0	2	2.00x
0	4	3.92x
0	8	7.23x
0	16	7.14x
1	2	1.97x
1	4	2.4x
1	8	3.92x
1	16	6.94x
2	2	1.69x
2	4	2.56x
2	8	4.23x
2	16	5.95x

The speedup is not linear in the number of cores used.

My hypothesis is as followed. Since there are only 8 cores in the CPU, when program runs in 16 thread setting, more than one thread will run on the same core. For view 0, all pixel has the same amount of calculation. Therefore all threads have same amount of calculation. 8 cores of CPU will have same amount of calculation. They start to do the calculation and end at the same time. So, whether it is 8 threads or 16 threads, all of the calculation are evenly distributed on 8 cores. There is no speed up from 8 threads to 16 threads.

However, in the case of view 1 and view 2. Each pixel has different amount of calculation. One thread can run after another thread ends. Therefore the more threads it has, cores can be utilized more fully. The speedup from 8 to 16 will go up.

1.3 How do your measurements explain the speedup graph you previously created?

View	Thread	Ave. time for one thread	View	Thread		View	Thread	
0	2	0.131	1	2	0.031	2	2	0.022
0	4	0.066	1	4	0.006	2	4	0.008
0	8	0.033	1	8	0.001	2	8	0.003
0	16	0.029	1	16	0.0001	2	16	0.0018

For view 0, the average time to finish one thread is the same for 8-thread and 16-thread setting. It confirms my hypothesis that every thread has same amount of calculation and 8 cores have been fully utilized and there is no improvement going from 8 threads to 16 threads. For view 1 and view 2, the time for each thread keeps decreasing, more threads utilize the cores more fully, since each thread has different amount of calculation.

1.4 Modify the mapping of work

```
void mandelbrotSerial2(
                      float x0, float y0, float x1, float y1,
                      int width, int height,
                      int startRow, int endRow,
                      int maxIterations,
                      int output[], int numThreads)
{

    float dx = (x1 - x0) / width;
    float dy = (y1 - y0) / height;

    for (int j = startRow; j < endRow; j+=numThreads) {
        for (int i = 0; i < width; ++i) {
            float x = x0 + i * dx;
            float y = y0 + j * dy;

            int index = (j * width + i);
            output[index] = mandel(x, y, maxIterations);
        }
    }
}
void* workerThreadStart(void* threadArgs) {
    // double s = CycleTimer::currentSeconds();
    WorkerArgs* args = static_cast<WorkerArgs*>(threadArgs);

    mandelbrotSerial2(args->x0, args->y0, args->x1, args->y1, args->width,
        args->height,
    args->threadId, args->height, args->maxIterations,
    args->output, args->numThreads);
    // fprintf(stdout,"Tread %d finished in %lf sec.", args->threadId,
        CycleTimer::currentSeconds()-s);
    return NULL;
}
```

Describe your approach

The image is separated into rows of pixels. Supposed there are N threads in total, the i-th thread will calculate for the $Nx+i (x=0,1,2..)$ row.

Comment on the difference in scaling behavior

View	Thread	SpeedUp	View	Thread	SpeedUp	View	Thread	SpeedUp
0	2	1.99x	1	2	1.96x	2	2	1.99x
0	4	3.89x	1	4	3.82x	2	4	3.82x
0	8	7.64x	1	8	7.61x	2	8	7.56x
0	16	7.61x	1	16	7.55x	2	16	7.42x

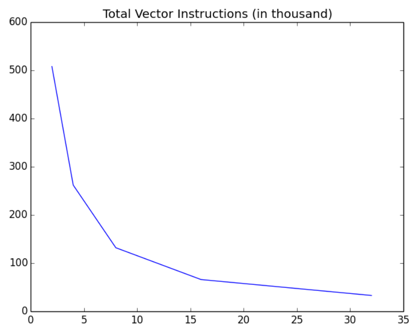
The speedup increases a lot from 4-thread to 8-thread, but actually decreases slightly from 8-thread to 16-thread. In this new strategy of allocating work, each thread has similar amount of calculation work. Therefore, going from 4 to 8, 8 cores of CPU can be utilize more fully. The increase in speedup is high. Whereas going from 8 to 16-thread, the utilization of each core doesn't improve much, because each thread has similar amount of work. Moreover, the overhead of switching thread makes the speedup decrease slightly.

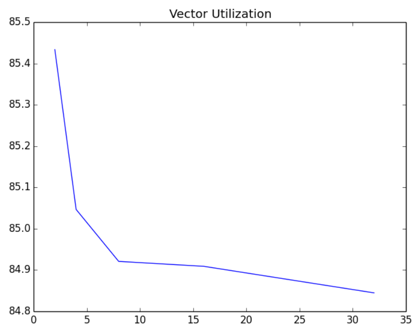
2 Problem 2: Vectorizing Code Using SIMD Intrinsics

2.1 Implement a vectorized version of clampedExpSerial()

```
void clampedExpVector(float* values, int* exponents, float* output, int N
    ) {
    // TODO: Implement your vectorized version of clampedExpSerial here
    //  ...
        __cmu418_vec_float x, xpower, res;
        __cmu418_vec_int y, e, isOneInt;
        __cmu418_mask isContinue, dataMask, tmp;

        // const value
        __cmu418_mask oneMask=_cmu418_init_ones(), zeroMask=
            _cmu418_init_ones(0);
        __cmu418_vec_int oneInt=_cmu418_vset_int(1), zeroInt=
            _cmu418_vset_int(0);
        __cmu418_vec_float vec418=_cmu418_vset_float(4.18f);


        for (int i=0;i<N+VECTOR_WIDTH;i+=VECTOR_WIDTH){
                dataMask=_cmu418_init_ones(N-i<VECTOR_WIDTH?N-i:
                    VECTOR_WIDTH);
                isContinue=dataMask;
                _cmu418_vload_float(x,values+i,dataMask);
                _cmu418_vset_float(res,1.0f,dataMask);
                _cmu418_vload_int(y,exponents+i,dataMask);
                _cmu418_vload_float(xpower,values+i,dataMask);

                while (_cmu418_cntbits(isContinue)>0){
                        // calculation
                        _cmu418_vbitand_int(isOneInt,y,oneInt,dataMask);
                        _cmu418_veq_int(tmp,isOneInt,oneInt,dataMask);
                        _cmu418_vmult_float(res,res,xpower,tmp);

                        _cmu418_vgt_float(tmp, res, vec418, dataMask);
                        _cmu418_vset_float(res,4.18f,tmp);
                        _cmu418_vmult_float(xpower,xpower,xpower,dataMask
                            );
                        _cmu418_vshiftright_int(y,y,oneInt,dataMask);
            tmp = _cmu418_mask_not(tmp);
            isContinue = _cmu418_mask_and(tmp,isContinue);
            _cmu418_vgt_int(tmp,y,zeroInt ,isContinue);
            isContinue = _cmu418_mask_and(tmp,isContinue);
        }
        _cmu418_vstore_float(output+i , res , dataMask);
    }
}
```

2.2 Record the resulting vector utilization





2.3 How much does the vector utilization change as W changes? Explain the reason for these changes and the degree of sensitivity the utilization has on the vector width.

The vector utilization is decreasing with respect to W. The utilization is decreasing because some elements in the vector is waiting for other elements to finish their calculation after they have finished the calculation. The wider the vector is, the more idle elements there are. Therefore the utilization is decreasing.

The utilization ratio is calculated by $\frac{actual\ work}{total\ work} = \frac{actual\ work}{actual\ work+idle\ work} = \frac{a}{a+i}$. When the width of vector going from 2 to 32, the i is increasing, a is not changing. The increased idle work relates to difference between the element with largest amount of calculation and other elements, which in practice, can be linear to the width of the vector. Therefore $utilization = \frac{a}{a+i}$; $i \propto \#thread$; $utilization \propto \frac{1}{\#thread}$

2.4 Extra: Implement a vectorized version of arraySumSerial()

```
float arraySumVector(float* values , int N) {
    // TODO: Implement your vectorized version here
    // ...
        float ans=0.f;
        float tmp[VECTOR_WIDTH];
        __cmu418_vec_float x, y;
        __cmu418_mask dataMask, oneHot, ones1, ones2;

        for (int i=0;i<N;i+=VECTOR_WIDTH){
                int endId=N-i<VECTOR_WIDTH?N-i:VECTOR_WIDTH;
                dataMask=_cmu418_init_ones(endId);
                _cmu418_vload_float(x,values+i,dataMask);

                while (endId>1){
                        if (endId%2==1){
                                // add 0 to pos endId; endId+=1
                                ones1=_cmu418_init_ones(endId+1);
                                ones2=_cmu418_init_ones(endId);
                                oneHot=_cmu418_mask_xor(ones1, ones2);
                                _cmu418_vset_float(x,0.f,oneHot);
                                endId+=1;
                        }
                        _cmu418_hadd_float(y,x);
                        _cmu418_interleave_float(x,y);
                        endId=endId/2;
                }
                ones1=_cmu418_init_ones(1);
                _cmu418_vstore_float(tmp,x,ones1);
                ans+=tmp[0];
        }

    return ans;
}
```

3 Problem 3: Using Instruction-level Parallelism in Fractal Generation

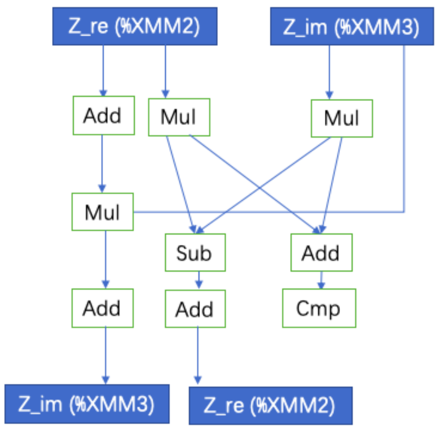
3.1 An annotated version of the assembly code

```
.L123:
        vmulss  %xmm2, %xmm2, %xmm4
        # store z_re*z_re into %xmm4

        vmulss  %xmm3, %xmm3, %xmm5
        # store z_im*z_im into %xmm5

        vaddss  %xmm5, %xmm4, %xmm6
        # store z_re*z_re+z_im*z_im into %xmm6

        vucomiss        .LC0(%rip), %xmm6
        # compare %xmm6 with 4.f

        ja      .L126
        # if %xmm6>4.f, break the loop

        vaddss  %xmm2, %xmm2, %xmm2
        # else: overwrite z_re with 2*z_re

        addl    $1, %eax
        # increment i with 1

        cmpl    %edi, %eax
        # compare i with cnt, Set condition code
        for jne below

        vmulss  %xmm3, %xmm2, %xmm3
        # overwirte z_im with 2*z_re*z_im

        vsubss  %xmm5, %xmm4, %xmm2
        # store z_re*z_re-z_im*z_im into %xmm2

        vaddss  %xmm3, %xmm1, %xmm3
        # update z_im to 2*z_re*z_im+c_im

        vaddss  %xmm2, %xmm0, %xmm2
        # update z_re to z_re*z_re-z_im*z_im+c_re

        jne     .L123
        # if i<cnt, loop again
```

Explain: How is the expression $2.f * z_{re} * z_{im}$ is implemented

$2 * z_{re} * z_{im}$ is implemented by first overwriting z_{re} with $2 * z_{re}$, and then overwriting z_{im} with $2 * z_{re} * z_{im}$.

3.2 Create a data-flow diagram



3.3 Determine the latency bound for function mandel ref. How close does the measured performance come to reaching this bound?

The critical path from data-flow is that:

$$\%xmm2- > add- > \%xmm2- > mul- > \%xmm3- > add- > \%xmm3$$

$$(\%xmm2- > mul)(\%xmm3- > mul)- > sub- > \%xmm2- > add- > \%xmm2$$

Following the write up, Add and Mul in unit0 and unit1 is 4 and add in unit 5 is 3. The final two "add" operations in the two path will compete for unit5. The latency bound is 11 cycles, as shown in the figure. The ref. performance in write-up is 9.75. Compiler could have done some optimization on the code.

3.4 What is the highest throughput bound? How close does the measured performance come to reaching this bound?

For FP Addition: 5 (ops/iteration) / 3 (ops/cycle) = 1.67 cycles/iteration

For FP Multiplication: 3 (ops/iteration) / 2 (ops/cycle) = 1.5 cycles/iteration

For Both: 1.67 cycles/iteration

By using SIMD, it reaches a peak performance of 3.85 cycles per iteration for vector width of 5.

3.5 Ideas for how a compiler could generate code that runs faster on this particular microarchitecture.

3.6 (Extra credit.) A demonstration of how the compiler could generate code that runs faster.

4 Problem 4: Parallel Fractal Generation Using ISPC

4.1 Part 1. A Few ISPC Basics

Compile and run the program mandelbrot.ispc. What is the maximum speedup you expect? Why might the number you observe be less than this ideal? Compare different views.

The maximum speedup I expected is 8x. The observed one is 5.07x speedup. Since each instance generated by ISPC could have different amount of computing work, the speedup is not 8x exactly. Also, the instances are run on a single core, where they might be competing for resources with each other.

View0 has the highest speed up since all the points reaches the maximum iteration. The utilization of vector is high. And the speedup for other views is lower.

4.2 Part 2. Combining instruction-level and SIMD parallelism

How much speedup does this two-way parallelism give over the regular ISPC version? Does it vary across different inputs (i.e., different –views)? When is it worth the effort?

View	ISPC	ISPC+parallel
0	5.07x	7.58x
1	4.75	4.37x
2	3.90x	4.27x
3	4.55x	5.07x
4	4.49x	5.57x
5	3.64x	4.14x
6	4.72x	6.07x

Different views have different speedup. View0, view6 have the most improvement. When the two pixel calculated in the two-way method has similar amount of calculation, the ISPC+parallel will have higher speedup. This is the case of view 0. So when different elements can have similar amount of calculation, the ISPC + parallel will worth it.

4.3 Part 3: ISPC Tasks

Run mandelbrot ispc. What speedup do you ob- serve on view 1? What is the speedup that does not partition that computation into tasks?

view1 has 7.17 speedup by using task", whereas 4.76 without task".

Modify the calculation of taskCount

```
uniform int taskCount = ((width+BLOCK_WIDTH)/BLOCK_WIDTH)*((height+
    BLOCK_HEIGHT)/BLOCK_HEIGHT);
```

What region sizes and shapes work best? Setting width to 1600, height to 1, can achieve 38x speedup. I prefer wide block, it is because it fetches memory faster. Arrays are store by rows in memory.

Extra Credit: what are differences between Pthread and ISPC task abstraction? Pthread has higher overhead in memory and computation. Each thread need space to store its sepcial information. However, task is more light weight. It uses multiply thread to conduct different tasks. Tasks are put into a pool.

If 10000 pthreads are run, the operating system could shut down or run very slow because of rescheduling these threads. But if 10000 tasks are run, they can run more smoothly, because they are line in a queue or pool, different threads would finally excute them.

5 Problem 5: Iterative Cubic Root

5.1 Build and run cuberoot

4.70x speedup from ISPC, 34.97x speedup from task ISPC. So the speedup 4.70x is becuase of SIMD parallelization. $34.97/4.70 = 7.44$x speedup is due to multi-core paralleleization.

5.2 Good data

Good data is 1.9, 0.1 etc. The speedup for ISPC is 6.19x. The speedup for task is 47.36x. The speedup due to task is 7.56x.

Points near 0, 2, -2 are all good points. They need more iterations to finished the calculation, therefore the parallel is useful to them.

5.3 Bad data

Bad point is setting one of the element to be 1.9, others to be 1.0. Since 1.9 will requires much more iterations than 1.0, most of the elements in task will be waiting for the 1.9 to finish its calculation. The power of task parallelism is not shown at all. Therefore the relative speedup will be every low.

The speedup for ISPC is 2.73x. The speedup for ISPC+task is 2.85x. The speedup due to task is 1.04x.