Assignment 1 - Performance Analysis on a Quad-Core CPU

Problem 1 - Parallel Fractal Generation Using Threads

1. 修改 `mandelbrot.cpp` 中的代码以实现并行计算
2. 修改 `workerThreadStart` 的代码以实现
3. 研究针对不同图像的每个线程耗时，并解释原因
4. 实现一个8x的并行程序

Q&A

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

我们针对 `view1` 进行测试

线程	耗时 /ms	加速比
1	74.321	1
2	37.048	2.01
3	24.824	2.99
4	18.942	3.92
5	16.844	4.41
6	14.119	5.26
7	12.197	6.093
8	10.648	6.979
9	17.736	4.19

这不是线性的，随着线程数增加，加速比增长速度逐渐下降。一直到9线程（本机CPU是8线程）时加速比降低。

按照理论，如果是线性关系，那么到达8线程的时候应该加速比接近8。考虑到我的本机CPU只有4核心，我不认为在5线程后每个线程性能是一致的。因此我们可以观察4线程时，可以注意到加速比为3.92，是比较接近4x加速的。

我个人认为，在5-8线程中，加速比增速下降的主要原因是cpu调度，cpu任务分配的耗时占了大头。

查阅了一些资料，是因为intel的超线程技术中，单线程性能不及一般线程，这也就解释了为什么5-8线程的时候加速比增速下滑了。

> To confirm (or disprove) your hypothesis, measure the amount of time each thread requires to complete its work by inserting timing code at the beginning and end of workerThreadStart(). How do your measurements explain the speedup graph you previously created?

在这个问题中，我给每个核心都开了性能模式，因此速度会快一点

```
❯ ./mandelbrot --view 1 --threads 4
[mandelbrot serial]:            [71.300] ms
Wrote image file mandelbrot-v1-serial.ppm
[thread 3]: [18.512] ms
[thread 2]: [18.527] ms
[thread 1]: [18.593] ms
[thread 0]: [18.725] ms
[thread 2]: [18.523] ms
[thread 1]: [18.584] ms
[thread 3]: [18.573] ms
[thread 0]: [18.591] ms
[thread 3]: [18.553] ms
[thread 2]: [18.577] ms
[thread 1]: [18.655] ms
[thread 0]: [18.670] ms
[mandelbrot thread]:            [18.669] ms
Wrote image file mandelbrot-v1-thread-4.ppm
++++                            (3.82x speedup from 4 threads)

❯ ./mandelbrot --view 2 --threads 4
[mandelbrot serial]:            [42.205] ms
Wrote image file mandelbrot-v2-serial.ppm
[thread 3]: [11.019] ms
[thread 2]: [11.066] ms
[thread 0]: [11.096] ms
[thread 1]: [11.224] ms
[thread 2]: [11.047] ms
[thread 3]: [11.029] ms
[thread 1]: [11.075] ms
[thread 0]: [11.124] ms
[thread 1]: [11.076] ms
[thread 2]: [11.236] ms
[thread 3]: [12.659] ms
[thread 0]: [12.844] ms
[mandelbrot thread]:            [11.173] ms
Wrote image file mandelbrot-v2-thread-4.ppm
++++                            (3.78x speedup from 4 threads)
```

其实不明白为何验证非线性结论要求我计算每个线程的耗时

但是我发现了一个新问题：我开了4 threads,但是无论怎么改都有12个输出

后面改了不同的线程，1threads有3个输出，2就有6个，3就有9个，神奇啊（大雾）

> Modify the mapping of work to threads to improve speedup to 8× on view 0 and almost 8× on views 1 and 2. In your writeup, describe your approach and report the final 16-thread 3 speedup obtained. Also comment on the difference in scaling behavior from 4 to 8 threads versus 8 to 16 threads.

就是让我们改程序，以让它可以实现8x速度，但是cmu的高端电脑似乎是8核心16线程的，我的老破小是4核心8线程的，应该只要实现4x速度吧（大雾）

但是似乎已经改不到更快了，3.92x已经是极限了（悲）

Code

```cpp
// Line 109-128
void *workerThreadStart(void *threadArgs) {
    WorkerArgs *args = static_cast<WorkerArgs *>(threadArgs);

    // TODO: Implement worker thread here.
    int threadNumRows = args->height / args->numThreads;
    if(args->height % args->numThreads != 0) {
        threadNumRows++;
    }
    // int threadStartRow = args->threadId * threadNumRows;
    // int threadTotalRows = threadNumRows;

    // mandelbrotSerial(args->x0, args->y0, args->x1, args->y1, args->width, args->height, threadStartRow, threadTotalRows, args->maxIterations, args->output);

    for(int i = 0; i < threadNumRows; i++) {
        int threadStartRow = i * args->numThreads + args->threadId;
        if(threadStartRow < args->height) {
            mandelbrotSerial (args->x0, args->y0, args->x1, args->y1, args->width, args->height, threadStartRow, 1, args->maxIterations, args->output);
        }
    }
    return NULL;
}

// Line 147 - 159
for (int i = 0; i < numThreads; i++) {
    args[i].threadId = i;
    // TODO: Set thread arguments here
    args[i].x0 = x0, args[i].y0 = y0;
    args[i].x1 = x1, args[i].y1 = y1;
    args[i].width = width;
    args[i].height = height;
    args[i].maxIterations = maxIterations;
    args[i].output = output;
    args[i].numThreads = numThreads;
}
```

Problem 2 - Vectorizing Code Using SIMD Intrinsics

1. 修改 `functions.cpp` 中的 `clampedExpVector()` 以实现并行快速幂
2. 修改 `functions.cpp` 中的 `arraySumVector()` 以实现并行数组求和

Q&A

> How much does the vector utilization change as W changes? Explain the reason for these changes and the degree of sensitivity the uti-lization has on the vector width. Explain how the total number of vector instructions varies with W .

我们运行参数为 `./vrun -s 10000`

Vector Width	Vector Utilization	Total Vector Instructions
2	92.980082%	383929
4	92.812667%	196495
8	92.777550%	98739
16	92.776768%	49378
32	92.778057%	24651

我们可以发现的，在误差范围内，当向量宽度逐渐变大时，我们向量利用率是逐步下降的，但是下降速率逐渐降低

利用率下降的主要原因应该是，代码中我对余数部分的数组线性处理，而当宽度变大时，线性处理的部分也会增加

Code

我一开始是直接把上面的快速幂改成向量的，只处理了除不尽的情况，然后当s 为奇数的时候会Fail

后面忘记看了什么资料发现别人的代码中是移走if(result > 4.18f)的代码到while结束，手动改了一下就过了

2023-03-31：重新回顾了一下这个代码，发现我们不需要线性处理，我们只需要设置一下掩码即可，修改后效率可以达到 `94.207317%` ，提升2%

下面这份是修改后的代码，修改前的话则是对余数部分线性处理

```
void clampedExpVector(float *values, int *exponents, float *output, int N) {
    // TODO: Implement your vectorized version of clampedExpSerial here
    __cmu418_vec_float x, result, xpower;
    __cmu418_vec_int y, tmp_y; // tmp_y stores y & 0x1

    __cmu418_vec_float exp = _cmu418_vset_float(4.18f);
    __cmu418_vec_int zero = _cmu418_vset_int(0);
    __cmu418_vec_int one = _cmu418_vset_int(1);

    __cmu418_mask maskAll, maskIsNegative, maskIsNotNegative, maskResultIsClamped;

    int cnt = N % VECTOR_WIDTH;
```

```
    // if(cnt) clampedExpSerial(values, exponents, output, cnt); // Serial Solve

    // Vector Solve
    for (int i = 0; i < N; i += VECTOR_WIDTH) {
        maskAll = (i == 0 && cnt) ? _cmu418_init_ones(cnt) : _cmu418_init_ones();
        maskIsNegative = _cmu418_init_ones(0);

        _cmu418_vload_float(x, values + i, maskAll);                // x = values[i];
        _cmu418_vset_float(result, 1.f, maskAll);                   // result = 1.f;
        _cmu418_vload_int(y, exponents + i, maskAll);               // y =
exponents[i];
        _cmu418_vmove_float(xpower, x, maskAll);                    // xpower = x;

        while(_cmu418_vgt_int(maskIsNegative, y, zero, maskAll),
_cmu418_cntbits(maskIsNegative)) // while(y > 0)
        {
            _cmu418_vbitand_int(tmp_y, y, one, maskAll);                    // tmp_y
= y & 0x1
            _cmu418_vgt_int(maskIsNegative, tmp_y, zero, maskAll);          //
if(tmp_y) {
            _cmu418_vmult_float(result, result, xpower, maskIsNegative);    //
result *= xpower
                                                                            // }
            _cmu418_vmult_float(xpower, xpower, xpower, maskAll);           // xpower
*= xpower
            _cmu418_vshiftright_int(y, y, one, maskAll);                    // y >>=
1
        }

        _cmu418_vgt_float(maskResultIsClamped, result, exp, maskAll);   //  if(result >
4.18f) {
        _cmu418_vmove_float(result, exp, maskResultIsClamped);          //      result =
4.18f
                                                                        //  }
        _cmu418_vstore_float(output + i, result, maskAll);              // output[i] =
result

        if(i == 0 && cnt) i -= VECTOR_WIDTH;
    }
}
```

这里有个未解之谜，关于Bonus的

```
float arraySumVector(float *values, int N) {
    // TODO: Implement your vectorized version here

    __cmu418_vec_float x, result;
    __cmu418_vec_float zero = _cmu418_vset_float(0.f);
    __cmu418_mask maskAll, maskIsNegative, maskIsNotNegative;

```

```
    float sum[VECTOR_WIDTH];
    float ans = 0.f;

    for(int i = 0; i < VECTOR_WIDTH; i++)
    {
        sum[i] = 0.f;
        // printf("sum[%d] = %f\n", i, sum[i]);
    }

    for(int i = 0; i < N; i += VECTOR_WIDTH)
    {
        maskAll = _cmu418_init_ones();
        maskIsNegative = _cmu418_init_ones(0);

        _cmu418_vload_float(x, values + i, maskAll);                    // x =
values[i];
        _cmu418_vadd_float(result, result, x, maskAll);                 // result +=
x;
        _cmu418_vstore_float(sum + (i % VECTOR_WIDTH), result, maskAll);    // sum[i] =
result;
    }

    for(int i = 0; i < VECTOR_WIDTH; i++) printf("sum[%d] = %f\n", i, sum[i]), ans +=
sum[i];

    return ans;
}
```

这是我的这份代码，可以注意到我有一个printf的输出，当我注释掉输出时，会输出如下

```
ARRAY SUM (bonus)
sum[0] = -1.010000
sum[1] = -1.007556
sum[2] = -1.005328
sum[3] = -1.000470
sum[4] = -1.006793
sum[5] = -1.003835
sum[6] = -25613815022341035519348437103411 2.000000
sum[7] = -1.000535
Expected -8.032826, got -256138150223410355193484371034112.000000
.@@@ Failed!!!
```

当我没有注释输出时，会输出如下

```
ARRAY SUM (bonus)
sum[0] = 0.000000
sum[1] = 0.000000
sum[2] = 0.000000
sum[3] = 0.000000
sum[4] = 0.000000
```

```
sum[5] = 0.000000
sum[6] = 0.000000
sum[7] = 0.000000
sum[0] = -1.000000
sum[1] = -1.007556
sum[2] = -1.015328
sum[3] = -1.000470
sum[4] = -1.006793
sum[5] = -1.003835
sum[6] = -1.008310
sum[7] = -1.000535
Expected -8.032826, got -8.042827
.@@@ Failed!!!
```

？？？多一个printf会有这么大的影响嘛，不过最终答案的确也是错的，不知道发生了什么。太玄学了！

看了一下 `CMU418intrin.h` ，我们肯定是要用上 `_cmu418_hadd_float()` 和 `_cmu418_interleave_float()` 的，因为这两个很突兀。第一个是实现[0 1 2 3] → [0+1 0+1 2+3 2+3]，不好描述，就是按pair相加，第二个是实现把一个数组的奇数下标移到前半部分，偶数下标移动到后半部分

很明显嘛，就是循环使用这两个函数，类似二分，每次都会折半，题目中说了VECTOR_WIDTH是2的倍数，那么我们可以按位处理。

然后同时还发现了这行代码 `template <typename T> struct __cmu418_vec { T value[VECTOR_WIDTH]; };` ，原来可以直接输出向量的某一个下标

那就好办了，不需要sum了

```
float arraySumVector(float *values, int N) {
    // TODO: Implement your vectorized version here

    __cmu418_vec_float x, result = _cmu418_vset_float(0.f);
    __cmu418_mask maskAll = _cmu418_init_ones();
    int cnt = VECTOR_WIDTH;

    for(int i = 0; i < N; i += VECTOR_WIDTH)
    {
        _cmu418_vload_float(x, values + i, maskAll);                    // x =
values[i];
        _cmu418_vadd_float(result, result, x, maskAll);                 // result +=
x;
    }

    while(cnt >>= 1)
    {
        _cmu418_hadd_float(result, result);
        _cmu418_interleave_float(result, result);
    }

    return result.value[0];
}
```

最终结果

```
❱ ./vrun -s 8
CLAMPED EXPONENT (required)
Results matched with answer!
****************** Printing Vector Unit Statistics *******************
Vector Width:              8
Total Vector Instructions: 82
Vector Utilization:        93.140244%
Utilized Vector Lanes:     611
Total Vector Lanes:        656
************************ Result Verification *************************
Passed!!!

ARRAY SUM (bonus)
Passed!!!
```

Problem 3 - Using Instruction-level Parallelism in Fractal Generation

1. 理解并注释 `mandelbrot_ref_loop.s` 中的代码

Q&A

> An annotated version of the assembly code for the main loop of mandel ref

我们可以在这里查询汇编指令：x86 and amd64 instruction reference

我们可以在这里查询一些其他资料：并发原理 – CPU原子性指令（一）

那么其实就是查询汇编的答案了，但是还是有点不会。

要知道一些基本信息，例如x86-64的16个64位寄存器

Name	Description
%xmmi	128位寄存器，i为编号
%rip	指向下一条要执行的指令
%eax	accumulator，加乘法指令缺省寄存器
%edi	destination indexm，目标索引寄存器

> An analysis of the latency bound of mandel ref and how the measured performance compares.

主要是提交一份Mandel Ref的延迟分析

这份代码中我觉得比较多的是xmm寄存器的操作，查看Assignments的介绍中有提及：U0可以执行加乘除、U1可以执行加乘、U5可以做浮点加法，但是只能使用x87浮点指令。U0/U1有4个时钟周期，U5有3个时钟周期。

但是我没有在The microarchitecture of Intel and AMD CPUs中对应页面找到解释（？）难道是更新了，似乎是在P155

这部分很薄弱，我的分析可能存在问题：

整个loop中涉及到了4次寄存器乘法，5次寄存器加法，还有2次jmp，latency至少是4 * 4 + 5 * 3 + 1 * 2 = 26 时钟周期

> Considering the floating-point operations, what is the highest throughput bound imposed by the functional units?

考虑浮点运算，功能单元所施加的最高吞吐量是多少

这个超纲了啊！！！

Code

以下给出这份代码的注释版本，但是我感觉没有写好

```
        # This is the inner loop of mandel_ref
        # Parameters are passed to the function as follows:
        #   %xmm0: c_re
        #   %xmm1: c_im
        #   %edi:  count
        # Before entering the loop, the function sets registers
    # to initialize local variables:
        #   %xmm2: z_re = c_re
        #   %xmm3: z_im = c_im
        #   %eax:  i = 0
.L123:
        vmulss  %xmm2, %xmm2, %xmm4         # xmm2 *= xmm4
        vmulss  %xmm3, %xmm3, %xmm5         # xmm3 *= xmm5
        vaddss  %xmm5, %xmm4, %xmm6         # xmm5 = xmm4 * xmm6
        vucomiss        .LC0(%rip), %xmm6     # Compare xmm1 and xmm6 and set the EFLAGS
flags accordingly.
        ja      .L126               # if larger, jump to .L126
        vaddss  %xmm2, %xmm2, %xmm2         # xmm2 += xmm2
        addl    $1, %eax                # count++
        cmpl    %edi, %eax                # Set condition codes for jne below
        vmulss  %xmm3, %xmm2, %xmm3         # xmm3 *= xmm2
        vsubss  %xmm5, %xmm4, %xmm2         # xmm5 = xmm4 - xmm2
        vaddss  %xmm3, %xmm1, %xmm3         # xmm3 += xmm1
        vaddss  %xmm2, %xmm0, %xmm2         # xmm2 += xmm0
        jne     .L123
```

Problem 4 - Parallel Fractal Generation Using ISPC

出了点小插曲，我的电脑中因为更新已经删除了 `sys/sysctl.h` ，不过似乎注释掉 `common/tasksys.h` 中的include就可以编译了，那导入这个库干嘛..

Problem 4, Part 1. A Few ISPC Basics

1. 编译程序，并提出自己认为的理论值

Q&A

> What is the maximum speedup you expect given what you know about these CPUs? Why might the number you observe be less than this ideal?

首先，既然是 8-wide AVX vector instructions，那理论加速比应当为8x，但是实际上我跑的理论加速比只有4.48x

我认为没有达到实际加速比是因为调度问题，毕竟不可能直接到8x，但是4.48x的差距还是太大了，说明不止是调度问题

查阅了网上 ~~的标准答案~~ 他人的想法， 负载不均衡的问题可能是比较正确的，我们实际上跑的这个图不是均匀的，那么就会造成有的快有的慢。

Problem 4, Part 2. Combining instruction-level and SIMD parallelism

Q&A

> 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?

这道题做崩了，反向加速了，但是可以注意到在视图不同的时候，反向加速效果也是不一样的，但是我感觉我的代码没有很大的问题，实在是不知道为啥反向加速了，难道是自己电脑CPU太古早了?

```
❯ ./mandelbrot_ispc -v 1
[mandelbrot serial]:            [183.684] ms
Wrote image file mandelbrot-1-serial.ppm
[mandelbrot ispc]:              [44.045] ms
Wrote image file mandelbrot-1-ispc.ppm
[mandelbrot ispc parallel]:             [66.698] ms
Wrote image file mandelbrot-1-ispc-par.ppm
                                (4.17x speedup from ISPC)
                                (2.75x speedup from ISPC+parallelism)
❯ ./mandelbrot_ispc -v 2
[mandelbrot serial]:            [107.956] ms
Wrote image file mandelbrot-2-serial.ppm
[mandelbrot ispc]:              [30.914] ms
Wrote image file mandelbrot-2-ispc.ppm
[mandelbrot ispc parallel]:             [43.129] ms
Wrote image file mandelbrot-2-ispc-par.ppm
```

```
                                (3.49x speedup from ISPC)
                                (2.50x speedup from ISPC+parallelism)
❱ ./mandelbrot_ispc -v 3
[mandelbrot serial]:            [260.036] ms
Wrote image file mandelbrot-3-serial.ppm
[mandelbrot ispc]:              [65.158] ms
Wrote image file mandelbrot-3-ispc.ppm
[mandelbrot ispc parallel]:             [89.083] ms
Wrote image file mandelbrot-3-ispc-par.ppm
                                (3.99x speedup from ISPC)
                                (2.92x speedup from ISPC+parallelism)
❱ ./mandelbrot_ispc -v 4
[mandelbrot serial]:            [257.352] ms
Wrote image file mandelbrot-4-serial.ppm
[mandelbrot ispc]:              [64.656] ms
Wrote image file mandelbrot-4-ispc.ppm
[mandelbrot ispc parallel]:             [88.640] ms
Wrote image file mandelbrot-4-ispc-par.ppm
                                (3.98x speedup from ISPC)
                                (2.90x speedup from ISPC+parallelism)
```

Code

```
export void mandelbrot_ispc_par2(uniform float x0, uniform float y0,
                                 uniform float x1, uniform float y1,
                                 uniform int width, uniform int height,
                                 uniform int maxIterations,
                                 uniform int output[]) {
    uniform float dx = (x1 - x0) / width;
    uniform float dy = (y1 - y0) / height;

    // TODO: Write ISPC code that will use function mandel_par2 to process
    // two rows on each pass.
    // You should use the foreach construct.
    // You should handle the case where the height is not a multiple
    // of 2.

    uniform int numRowsPerPass = (height + 1) / 2;

    foreach (i = 0 ... numRowsPerPass, j = 0 ... width) {
        int row0 = i * 2;
        int row1 = min(row0 + 1, height - 1);

        float c_re0 = x0 + j * dx;
        float c_im0 = y0 + row0 * dy;
        float c_re1 = x0 + j * dx;
        float c_im1 = y0 + row1 * dy;
```

```
        mandel_par2(c_re0, c_im0, c_re1, c_im1, maxIterations, output + row0 * width + j, output + row1 * width + j);
    }
}
```

Problem 4, Part 3: ISPC Tasks

1. 尝试不同的BLOCK大小与TASK数量，以超过20x加速

最终是 `20.99x` 加速，参数是

```
#define BLOCK_WIDTH 400
#define BLOCK_HEIGHT 200
uniform int taskCount = 16;
```

不过这个20.99x也蛮极限的，可能原因是CPU是4核心8线程的，16task其实是有点高了

有一个问题是说

Problem 5 - Iterative Cubic Root

1. 运行迭代立方根程序，看结果
2. 修改 `data.cpp` 中的内容，调优

```
❯ ./cuberoot
[cuberoot serial]:              [3658.882] ms
[cuberoot ispc]:                [884.804] ms
[cuberoot task ispc]:   [134.455] ms
                                (4.14x speedup from ISPC)
                                (27.21x speedup from task ISPC)
```

Q&A

> Modify the function initGood/initBad() in the file data.cpp to generate data that will yield a very high relative speedup of the ISPC implementations.
>
> Does your modification improve SIMD speedup? Does it improve multi-core speedup? Please explain why.

我最多可以跑到40x左右的加速，这时候 `value[i] = 1.9999999` ，似乎可以无限逼近2

```
❯ ./cuberoot -d g
[cuberoot serial]:              [7345.921] ms
[cuberoot ispc]:                [1190.029] ms
[cuberoot task ispc]:   [178.646] ms
                                (6.17x speedup from ISPC)
                                (41.12x speedup from task ISPC)
```

最慢可以达到2.3x的加速，这时候 `value[i] = 1.0`

```
[cuberoot serial]:               [80.059] ms
[cuberoot ispc]:                 [43.914] ms
[cuberoot task ispc]:    [34.633] ms
                                 (1.82x speedup from ISPC)
                                 (2.31x speedup from task ISPC)
```

实际上修改为1就可以实现最慢了，-1应该也可以，可以观察给的图形，在正负1.0处迭代最少，在无限逼近2的时候迭代次数徒增

要解释的话，我们可以说明ispc可以更好的处理数学计算？加速数学计算的过程吧。

这个问题5似乎比前面几个来的简单多啊。。这个看起来得放在第一问（）