CS149 Assignment 1: Performance Analysis on a Quad-Core CPU

Program 1: Parallel Fractal Generation Using Threads (20 points)

目标：使用多线程加速分形图案绘制

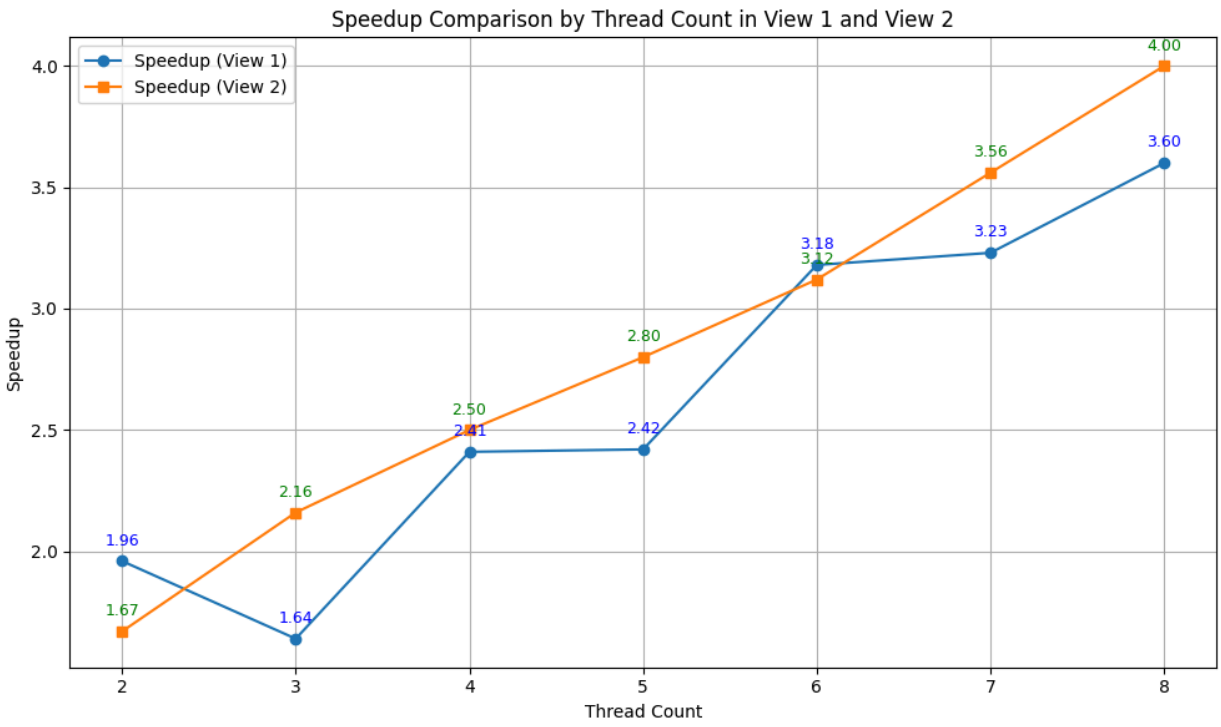
实现简单多线程版本

无需深入研究分形图案相关细节，先按一般思路将图案水平分为多个块分配给多个线程

由于线程数不一定能整除图案高度(假设每线程需要处理的行数为`baseRows`，多余行数量为`remainRows`)，因此需要处理边界情况，有如下两种方案：

- 每线程按`baseRows`进行分配，多余的行全部分配给最后一个线程
- 每线程按`baseRows`进行分配，多余的行从线程0开始每线程分配1个

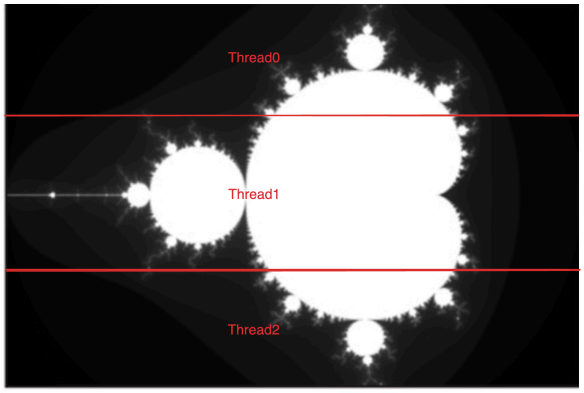
加速比分析



可以发现对于view1的图像，3线程运行时会出现奇怪的加速比下降，对比2线程和3线程的运行数据

```
Thread 2 time: 83.439 ms
Thread 0 time: 83.503 ms
Thread 1 time: 255.239 ms
```

图案中白色部分计算量更大，可以发现线程1的工作量远大于其余两线程，工作量不平衡



View 1

优化方案

为了使得每线程的工作量尽量平衡，应该试图将高工作量的白色区域平均分配个每个线程，如此应从分块分配转为交替分配，即先将图案分配为水平的一系列比较小的块，然后从上至下交替分配给每个线程，从平均意义上平衡工作量

> 一开始想的确实是分成小块再分配，因为这时候突然想到了缓存，怕缓存读了很多但却没用上，后来发现没必要图像一行是1200个int，缓存行一次是读64byte，所以单纯每次"分配"一行就行了，后面引入新函数也避免了从原线程函数频繁调用的问题(一行一调用还是有一定开销)

如共4个线程，线程0负责第0,4,8行.....

为此引入一个新函数`mandelbrotSerialStep`，原线程根据args调用，传入线程id作为起始行，线程数量作为间隔step

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

    for (int j = startRow; j < height; j+=step) {
        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);
        }
    }
}

---------------------------------------------
// mandelbrotThread.cpp
void workerThreadStart(WorkerArgs * const args) {
```

```
    // TODO FOR CS149 STUDENTS: Implement the body of the worker
    // thread here. Each thread should make a call to mandelbrotSerial()
    // to compute a part of the output image.  For example, in a
    // program that uses two threads, thread 0 could compute the top
    // half of the image and thread 1 could compute the bottom half.

    double startTime = CycleTimer::currentSeconds();
    // 由线程id决定从哪一行开始，间隔step由总线程数量决定，其余参数不变
    mandelbrotSerialStep(args->x0, args->y0, args->x1, args->y1,
                         args->width, args->height,
                         args->threadId, args->numThreads, args->maxIterations, args->ou
    double endTime = CycleTimer::currentSeconds();
    printf("[thread %d]:\t\t[%.3f] ms\n", args->threadId, (endTime-startTime)*1000
}
```

```
--------------------------------------
[thread 0]:             [33.590] ms
[thread 4]:             [33.670] ms
[thread 5]:             [33.713] ms
[thread 3]:             [33.745] ms
[thread 1]:             [33.760] ms
[thread 2]:             [33.811] ms
[thread 6]:             [33.799] ms
[thread 7]:             [33.785] ms
--------------------------------------
[mandelbrot thread]:            [33.863] ms
Wrote image file mandelbrot-thread.ppm
                                (6.86x speedup from 8 threads)
```

修正后达到6.86加速比，不知道为什么就是上不了7？🤷

最大加速比

对于4核8线程，线程数增加到16后，加速比几乎保持不变或略有下降，因为8线程已经使得处理器充分使用，过多的线程反而带来额外的切换开销

Program 2: Vectorizing Code Using SIMD Intrinsics (20 points)

要求使用CS149's "fake vector intrinsics"来将示例的`clampedExpSerial`转为SIMD程序，基本上就是一对一转换，需要注意的是通过mask掩码来进行控制

```
void clampedExpVector(float* values, int* exponents, float* output, int N) {

  //
  // CS149 STUDENTS TODO: Implement your vectorized version of
  // clampedExpSerial() here.
  //
  // Your solution should work for any value of
  // N and VECTOR_WIDTH, not just when VECTOR_WIDTH divides N
  //

  __cs149_vec<float> result, base, upperLimit;
  __cs149_vec<int> exp, zero, one;
  __cs149_mask maskAll, maskIsZeroExp, maskIsValid, maskNotZeroExp, maskVa

  upperLimit = _cs149_vset_float(9.999999f); // 设置上限常量

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

    maskAll = _cs149_init_ones(); // 所有lane有效
    maskIsZeroExp = _cs149_init_ones(0); // 初始化为全0

    zero = _cs149_vset_int(0); // int向量全0
    one = _cs149_vset_int(1); // int向量全1

    // 标明有效的lane
    if (i + VECTOR_WIDTH >= N) {
      maskIsValid = _cs149_init_ones(N - i); // 处理最后一组不足VECTOR_WIDTH的情
    } else {
      maskIsValid = _cs149_init_ones(); // 全部有效
    }
```

```
    addUserLog("start load base");

    _cs149_vset_float(result, 1.f, maskIsValid); // result = 1.f，初始化输出
    _cs149_vload_float(base, values+i, maskIsValid); // 加载输入base

    // load valid exp elements
    _cs149_vload_int(exp, exponents+i, maskIsValid); // 加载输入exp

    _cs149_veq_int(maskIsZeroExp, exp, zero, maskIsValid); // 判断exp==0的mask

    // _cs149_vset_float(result, 1.f, maskIsZeroExp);

    maskNotZeroExp = _cs149_mask_not(maskIsZeroExp); // 注意有效位
    maskNotZeroExp = _cs149_mask_and(maskNotZeroExp, maskIsValid); // 只保留

    // 计算幂，exp>0时不断累乘
    while(true) {
      if (_cs149_cntbits(maskNotZeroExp) == 0) // if exp == 0
        break;
      _cs149_vmult_float(result, result, base, maskNotZeroExp); // result *= base
      _cs149_vsub_int(exp, exp, one, maskNotZeroExp); // exp--

      addUserLog("if 0 < exp");
      _cs149_vlt_int(maskNotZeroExp, zero, exp, maskNotZeroExp); // 0 < exp
    }

    // 检查是否超过上限
    _cs149_vlt_float(maskValueOverflow, upperLimit, result, maskIsValid); // 9.999

    _cs149_vset_float(result, 9.999999f, maskValueOverflow); // result =9.999999f

    _cs149_vstore_float(output + i, result, maskIsValid); // 写回结果
  }
}
```

利用率测试

VECTOR_WIDTH	Vector Utilization
2	82.0%
4	74.7%
6	72.4%
8	70.9%

因为是指数运算，迭代次数根据指数而不同，`VECTOR_WIDTH`越长，出现不同指数的概率越大，因此`VECTOR_WIDTH`越长利用率越低(后续只有最大指数在进行迭代)

数组求和的SIMD实现

```
float arraySumVector(float* values, int N) {
  __cs149_vec<float> result = _cs149_vset_float(0.f);
  __cs149_vec<float> tmp; // 暂存数组数据至vector中
  __cs149_mask maskAll;

  maskAll = _cs149_init_ones();

  // O(N / VECTOR_WIDTH)
  for (int i=0; i<N; i+=VECTOR_WIDTH) {
    _cs149_vload_float(tmp, values+i, maskAll);

    _cs149_vadd_float(result, result, tmp, maskAll);
  }

  // O(log(VECTOR_WIDTH)) 效率完成vector的元素求和以得到最终结果
  int round = log2(VECTOR_WIDTH);
  while(round--) {
    _cs149_hadd_float(result, result);
    _cs149_interleave_float(result, result);
  }
   return result.value[0];
}
```

`_cs149_hadd_float`：将向量中每对相邻元素(0和1,2和3,4和5.....)互相相加，如[0 1 2 3] → [0+1 0+1 2+3 2+3]，如此可以通过一条指令完成部分求和，并使得向量的有效长度减半

`_cs149_interleave_float`：将向量中所有偶数索引元素移动到前半，奇数索引元素到后半，如[0 1 2 3 4 5 6 7] → [0 2 4 6 1 3 5 7]，结合上述`hadd`指令可以将有效元素整合在一起

即每次`hadd`先部分(相邻)求和然后`interleave`将有效的求和元素前移，这等价于一次折半的操作，因此所需时间为`log(VECTOR_WIDTH)`

例如对向量[1 2 3 4 5 6 7 8]进行求和：

```
初始:  [1 2 3 4 5 6 7 8]

第1轮:
hadd:   [1+2 1+2 3+4 3+4 5+6 5+6 7+8 7+8]
     = [3   3   7   7   11  11  15  15]
inter:  [3 7 11 15 3 7 11 15]

第2轮:
hadd:   [3+7 3+7 11+15 11+15 3+7 3+7 11+15 11+15]
     = [10  10  26    26    10  10  26    26]
inter:  [10 26 10 26 10 26 10 26]

第3轮:
hadd:   [10+26 10+26 10+26 10+26 10+26 10+26 10+26 10+26]
     = [36    36    36    36    36    36    36    36]
inter:  [36 36 36 36 36 36 36 36]

最终取result[0] = 36，即为数组和
```

Program 3: Parallel Fractal Generation Using ISPC (20 points)

Part 1 ISPC加速比分析

目标加速比应该是8，ISPC View1的加速比在4.7左右，View 2的加速比在4.2左右

首先加速比小于8的原因很明显，每8个向量元素中肯定会存在计算轮次不同的情况(黑/白)，造成最后仅有部分通道(lane)使用

View 1比View 2像素更集中(View 2经常黑色中突然来几个白色像素之类)，使得View 1的向量利用率更高，因而有更高的加速比

Part 2 ISPC Tasks加速比分析

1. 参数 `--tasks` 运行 `mandelbrot_ispc` 加速比为9.15，近似于单纯ISPC的2倍
2. 16个任务数量即可达到最佳加速比大致在27～30之间，由于每个任务在不同的核心上处理，考虑4核8线程应至少需要8个任务，又考虑同Prog1中的工作量平衡问题，因此任务数量要稍微多些以将工作量尽量平均化，好比8个任务中中间部分的线程计算量大，而此时其他线程已计算完毕，相当于闲置状态，实践中超过16个任务加速比无明显提升，反而存在切换开销问题

```
// 8 task
[mandelbrot serial]:            [214.416] ms
Wrote image file mandelbrot-serial.ppm
[mandelbrot ispc]:              [45.301] ms
Wrote image file mandelbrot-ispc.ppm
[mandelbrot multicore ispc]:    [12.167] ms
Wrote image file mandelbrot-task-ispc.ppm
                                (4.73x speedup from ISPC)
                                (17.62x speedup from task ISPC)

// 16 task
[mandelbrot serial]:            [213.126] ms
Wrote image file mandelbrot-serial.ppm
[mandelbrot ispc]:              [44.891] ms
Wrote image file mandelbrot-ispc.ppm
[mandelbrot multicore ispc]:    [7.385] ms
Wrote image file mandelbrot-task-ispc.ppm
                                (4.75x speedup from ISPC)
                                (28.86x speedup from task ISPC)
```

3. 貌似就跟线程和线程池的区别一样，ISPC任务是在线程池上调度的，每次创建任务不过是添加在了任务队列中，实际由ISPC控制的工作线程(线程池)取任务并执行，极大减少了线程创建和切换等开销

Program 4: Iterative `sqrt` (15 points)

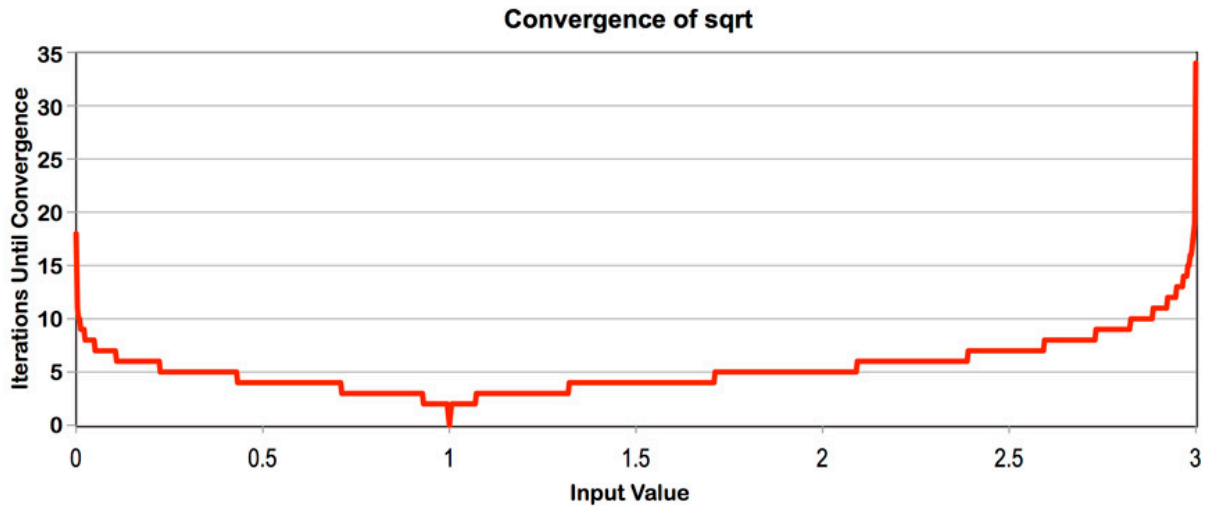
TODO

加速比分析

```
[sqrt serial]:      [740.830] ms
[sqrt ispc]:        [181.252] ms
[sqrt task ispc]:    [27.972] ms
                    (4.09x speedup from ISPC)
                    (26.48x speedup from task ISPC)
```

SIMD并行加速比为4.09x，多核加速比为6.47x(26➗4)

最大化加速比



由图可知不同的输入在sqrt的收敛次数会有很大差异，这正是每个8-lane向量中工作量不平衡的原因。为了确保工作量均衡，首先应将所有输入设置为同一数值

选哪个数值作为同一数值呢？考虑选择计算量最大数值(接近3)，为此可以使得并行计算时间所占比例更大，降低了串行和其他开销的比例，从而最大化加速比

我选择了`2.9999995f`，2.9999999f时间太长算不出来了

```
[sqrt serial]:        [4496.058] ms
[sqrt ispc]:          [673.690] ms
[sqrt task ispc]:      [108.098] ms
                      (6.67x speedup from ISPC)
                      (41.59x speedup from task ISPC)
```

最小化加速比

同上想达到最小加速比，要让工作量最不平衡，因此选择每8个位置放一个2.9999，其余均为1，结果如下

```
if (i % 8 == 0) values[i] = 2.9999f;
else  values[i] = 1.f;

-----------------------------------------
[sqrt serial]:        [381.773] ms
[sqrt ispc]:          [469.111] ms
[sqrt task ispc]:      [71.443] ms
                      (0.81x speedup from ISPC)
                      (5.34x speedup from task ISPC)
```

加速比小于1的原因：当为1的元素计算完成时，需通过mask屏蔽lane，因为计算1的计算量过小，使得这些SIMD的额外开销占比过大，加速比因此还不如串行运行

Program 5: BLAS `saxpy` (10 points)

```
[saxpy serial]:       [10.595] ms     [28.130] GB/s   [3.775] GFLOPS
[saxpy ispc]:         [10.076] ms     [29.578] GB/s   [3.970] GFLOPS
[saxpy task ispc]:     [5.467] ms      [54.509] GB/s   [7.316] GFLOPS
                      (1.84x speedup from use of tasks)
```

```
// main.cpp
const unsigned int N = 20 * 1000 * 1000; // 20 M element vectors (~80 MB)
```

```
const unsigned int TOTAL_BYTES = 4 * N * sizeof(float);
const unsigned int TOTAL_FLOPS = 2 * N;

printf("[saxpy ispc]:\t\t[%.3f] ms\t[%.3f] GB/s\t[%.3f] GFLOPS\n",
        minISPC * 1000,
        toBW(TOTAL_BYTES, minISPC),
        toGFLOPS(TOTAL_FLOPS, minISPC));
```

GFLOPS: 即每秒执行的浮点运算次数，本程序对于一个元素需执行`result[i] = scale * X[i] + Y[i]`，因此一个元素对应于2次浮点运算

加速比分析

从串行到ISPC几乎没有任何的性能提升，开了64个task的ispc也不过是接近2倍的加速比

对于串行和无task ISPC，浮点运算次数几乎没有很大的提升(考虑理想情况应该能到8倍)，合理怀疑瓶颈出现在带宽上

而64个task的ispc考虑在4c8t上执行，参考Prog4应该又有几倍的提升，但是仅有2倍，且带宽处于系统内存带宽上限(10-50GB/s)，GFLOPS远低于i5-1135g7的18.87GFLOPS

综上确认程序加速比的瓶颈是带宽，对应于Lecture 3中讲解带宽的经典一个指令对应于4个读写

为什么是4个读写？

为了写回result[i]，需先将result[i]对应的缓存行读入缓存，加上读取x[i]和y[i]，共4个读写

Program 6: Making K-Means Faster (15 points)

data.dat可通过main.cpp中的注释获取

在程序中加入计时代码后各部分时间如下所示

```
Reading data.dat...
Running K-means with: M=1000000, N=100, K=3, epsilon=0.100000
K-Means converged after 50 iterations
Total times:
```

```
  Assignment: 5369.852625 ms
  Centroid:   631.437083 ms
  Cost:       1246.556083 ms
  Total:      7247.845792 ms
[Total Time]: 7247.909 ms
```

题目要求加速比≥2.1，因此目标总时间3797ms，如此只能从优化assignment部分入手，观察代码发现，`computeAssignments`中与`m`相关的两个循环中，每一轮循环之间都是互相独立的，因此参考prog1，将任务平均分后分配个每个线程，代码如下

```
// 按线程ID得到赋值的区间并并行处理
void workerThreadStart(WorkerArgs *const args, int threadId, double *minDist) {
  int groupSize = args->M / args->numThreads; // 8个线程下是整除
  int startPos = threadId * groupSize;
  int endPos = startPos + groupSize;

  for(int m = startPos; m < endPos; m++) {
    minDist[m] = 1e30;
    args->clusterAssignments[m] = -1;
  }

  // Assign datapoints to closest centroids
  for (int k = args->start; k < args->end; k++) {
    for (int m = startPos; m < endPos; m++) { // 此处循环均独立
      double d = dist(&args->data[m * args->N], // dist调用
                      &args->clusterCentroids[k * args->N], args->N);
      if (d < minDist[m]) {
        minDist[m] = d;
        args->clusterAssignments[m] = k;
      }
    }
  }
}

void computeAssignments(WorkerArgs *const args) {
  double *minDist = new double[args->M]; // 或许可以每个线程创建单独的，没继续
```

```
  std::thread workers[MAX_THREADS];

  for (int i = 1; i < args->numThreads; i++) {
    workers[i] = std::thread(workerThreadStart, args, i, minDist);
  }

  workerThreadStart(args, 0, minDist);

  for (int i = 1; i < args->numThreads; i++) {
    workers[i].join();
  }

  free(minDist);
}
```

```
Reading data.dat...
Running K-means with: M=1000000, N=100, K=3, epsilon=0.100000
K-Means converged after 50 iterations
Total times:
  Assignment: 822.432667 ms
  Centroid:   640.906292 ms
  Cost:       1290.532083 ms
  Total:      2753.871042 ms
[Total Time]: 2753.920 ms
```

多线程化后的代码总时间为2753ms，加速比为2.63