15-418/618, Spring 2022
15-418/6182022 年春天

Assignment 1
作业 1

Exploring Multi-Core, Instruction-Level, and SIMD Parallelism
探索多核、指令级和 SIMD 并行性

Event 事件	Registered students 注册学生	Waitlist students 候补名单学生
Assigned: 分配:	Mon., Jan. 17 1 月 17 日星期一	Mon., Jan. 17 1 月 17 日星期一
Due: 截止日期:	Mon., Jan. 31, 11:59 pm Mon. 1 月 31 日 11:59 pm	Tue., Jan. 25, 11:59 pm 1 月 25 日，星期二，11:59 pm
Last day to handin: 最后一天交稿:	Thu., Feb. 3, 11:59 pm 星期四，2 月 3 日，11:59 pm	Tue., Jan. 25, 11:59 pm 1 月 25 日，星期二，11:59 pm

Overview
概述

In this assignment you will modify and experiment with code designed to exploit the three main forms of parallelism available on modern processors: the multiple cores that can execute programs independently, the multiple functional units that can operate in parallel, and the SIMD vector units that allow each processor to perform some of its arithmetic and memory operations on vectors of data.
在这次作业中，你们将修改和实验用于开发现代处理器上可用的三种主要并行形式的代码: 能够独立执行程序的多个核心，能够并行运行的多个功能单元，以及允许每个处理器对数据向量执行一些算术和内存操作的 SIMD 向量单元。

You will also gain experience measuring and reasoning about the performance of parallel programs, a chal-lenging, but important, skill you will use throughout this class. This assignment involves only a small amount of programming, but a lot of analysis!
你还将获得测量和推理并行程序性能的经验，这是一项具有挑战性但很重要的技能，你将在本课程中使用。这个作业只涉及少量的编程，但是有很多的分析！

This is an individual project. All handins are electronic. Your submission will consist of the code files you have modified, as well as a single document reporting your findings on the 5 problems described below.

You may use any document preparation system you choose, but the final result must be stored as a single file in PDF format, named report.pdf. Make sure the report includes your name and Andrew Id. More details on how to submit this information is provided at the end of this document.
这是一个单独的项目。所有的手都是电子的。你的提交将包括你修改过的代码文件，以及一个单独的文档来报告你对下面描述的 5 个问题的发现。你可以使用任何你选择的文档准备系统，但是最终的结果必须以 PDF 格式存储，命名为 report.PDF。确保报告包括你的名字和安德鲁 Id。关于如何提交这些信息的更多细节将在文档的最后提供。

Before you begin, please take the time to review the course policy on academic integrity at:
在开始之前，请花点时间阅读有关学术诚信的课程政策:

http://www.cs.cmu.edu/~418/academicintegrity.html
Http://www.cs.cmu.edu/418/academicintegrity. html

Getting started
开始

You will need to run code on the machines in the Gates cluster for this assignment. Host names for these machines are ghcX.ghc.andrew.cmu.edu, where X is between 47 and 86. Each of these machines has an eight-core, 3.0 GHz Intel Core i7 processor (although dynamic frequency scaling can take them
您将需要在 Gates 集群中的计算机上运行此任务的代码。这些机器的主机名是 ghcX.ghc.andrew.cmu.edu，x 在 47 到 86 之间。每台机器都有一个 8 核的 3.0 GHz Intel 酷睿 i7 处理器(尽管动态时钟频率调整可以让它们运行)

to 4.7 GHz when the chip decides it is useful and possible to do so). Each core can execute AVX2 vec-tor instructions, supporting simultaneous execution of the same operation on multiple data values (8 in the case of single-precision data). For the curious, a complete specification for this CPU can be found at https://ark.intel.com/content/www/us/en/ark/products/191792/intel-core-i7-9700-processor-1
当芯片认为有用并且可以这样做的时候，可以调整到 4.7 GHz)。每个核心可以执行 avx2 矢量指令，支持在多个数据值上同时执行相同的操作(单精度数据为 8)。有趣的是，这个 CPU 的完整规范可以在 https://ark.intel. com/content/www/us/en/ark/products/191792/intel-core-i7-9700-processor-1 中找到

You can log into these machines in the cluster, or you can reach them via ssh.
您可以登录集群中的这些机器，或者通过 ssh 访问它们。

We will grade your analysis of code run on the Gates machines; however for your own interest, you may also want to run these programs on other machines. To do this, you will first need to install the Intel SPMD Program Compiler (ISPC) available at: ispc.github.io/. Feel free to include your findings from running code on other machines in your report as well, just be very clear in your report to describe the machine(s) you used.
我们将为您在 Gates 机器上运行的代码分析打分；然而，为了您自己的利益，您也可能希望在其他机器上运行这些程序。要做到这一点，你首先需要安装英特尔 SPMD 程序编译器(ISPC)：ISPC.github.io/。您可以将在其他机器上运行代码的结果也包括在报告中，只要在报告中非常清楚地描述您使用的机器即可。

ISPC is needed to compile many of the programs used in this assignment. ISPC is currently installed on the Gates machines in the directory /usr/local/depot/ispc/bin/. You will need to add this directory to your system path.
需要 ISPC 来编译本作业中使用的许多程序。ISPC 目前安装在/usr/local/depot/ISPC/bin/目录下的 Gates 机器上。你需要将这个目录添加到你的系统路径中。

We will distribute the assignment starter code through the website:
我们将通过网站发布作业启动代码:

```
git clone https://github.com/cmu15418s22/Assignment-1.git
Git 克隆 https://github. com/cmu15418s22/assignment-1.git
```

1 Problem 1: Parallel Fractal Generation Using Pthreads (15 points)
问题 1: 使用 Pthreads 的并行分形生成(15 点)

Build and run the code in the prob1_mandelbrot_threads directory of the Assignment 1 code base. This program produces the image file mandelbrot-vV -serial.ppm, where V is the view index. This image is a visualization of a famous set of complex numbers called the Mandelbrot set. As you can see in the images below, the result is a familiar and beautiful fractal. Each pixel in the image corresponds to a value in the complex plane, and the brightness of each pixel is proportional to the computational cost of determining whether the value is contained in the Mandelbrot set—white pixels required the maximum
构建并运行 Assignment 1 代码库的 prob1 _ mandelbrot _ threads 目录中的代码。这个程序生成图像文件 mandelbrot-vV-serial.ppm，其中 v 是视图索引。这张图片是一组著名的复数的可视化集合，叫做 Mandelbrot 集合。正如你在下面的图片中看到的，结果是一个熟悉而美丽的分形。图像中的每个像素对应

于复平面上的一个值，每个像素的亮度与确定该值是否包含在 Mandelbrot 集合中的计算成本成正比——白色像素需要最大值

(256)number of iterations, dark ones only a few iterations, and colored pixels were somewhere in between. (See function mandel() defined in mandelbrot.cpp.) You can learn more about the definition of the Mandelbrot set at en.wikipedia.org/wiki/Mandelbrot set.

迭代次数，深色的迭代次数只有几次，彩色像素介于两者之间。(参见 mandelbrot.cpp 中定义的函数 mandel ()你可以在 en.wikipedia. org/wiki/Mandelbrot set 了解更多关于 Mandelbrot 集的定义。

Use the command option "--view V " for V between 0 and 6 to get the different images. You can click the links below to see the different images on a browser. Take the time to do this—the images are quite striking. (View 0 is not shown—it is all white.)

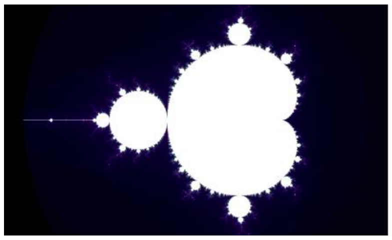
在 0 到 6 之间使用"—— viewv"命令选项来获得不同的图像。你可以点击下面的链接在浏览器上查看不同的图片。花点时间去做吧——这些图片非常引人注目。(视图 0 未显示——全部为白色

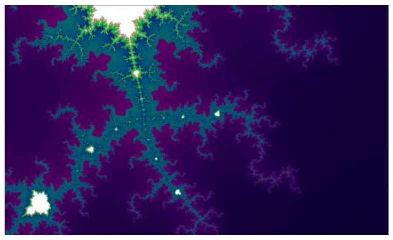
View 1 View 2 View 3

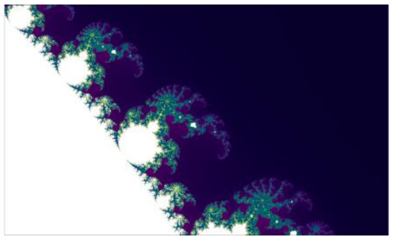
视图 1 视图 2 视图 3

1× magnification 66× magnification 50× magnification

1 倍放大倍数 66 倍放大倍数 50 倍

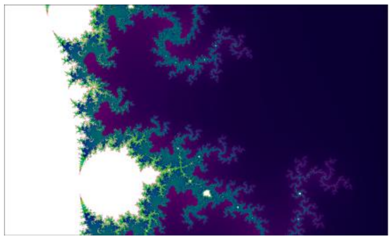
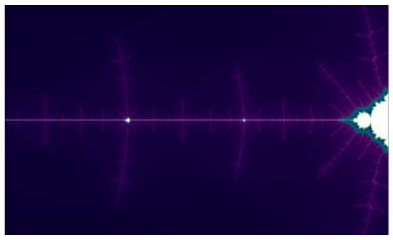
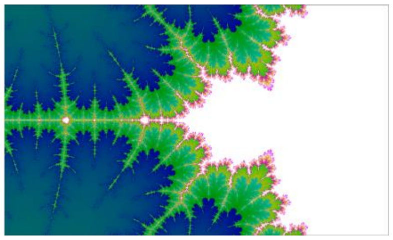






View 4 View 5 View 6
视图 4 视图 5 视图 6

50× magnification 50× magnification 500× magnification
50 倍放大 50 倍放大 500 倍放大

Your job is to parallelize the computation of the images using Pthreads. The command-line option “--threads T ” specifies that the computation is to be partitioned over T threads. In function mandelbrotThread(),
你的工作是使用 Pthreads 对图像进行并行计算。命令行选项“—— threads t”指定计算要在 t 线程上进行分区。在函数 mandelbrotThread ()中,

located in mandelbrot.cpp, the main application thread creates T −1 additional thread using pthread_create(). It waits for these threads to complete using pthread_join(). Currently, neither the launched threads
位于 mandelbrot.cpp 中的主应用程序线程使用 pthread _ create ()创建 t-1 附加线程。它等待这些线程使用 pthread _ join ()完成。当前，启动的线程

nor the main thread do any computation, and so the program generates an error message. You should add code to the workerThreadStart() function to accomplish this task. You will not need to use of any other Pthread API calls in this assignment.
也没有主线程进行任何计算，因此程序生成了一条错误消息。你应该在 workerThreadStart ()函数中添加代码来完成这个任务。在这个任务中，你不需要使用任何其他的 Pthread API 调用。

What you need to do:
你需要做的:

1. Modify the code in mandelbrot.cpp to parallelize the Mandelbrot generation using two cores. Specifically, compute the top half of the image in thread 0, and the bottom half of the image in thread 1. This type of problem decomposition is referred to as spatial decomposition since different spatial regions of the image are computed by different processors.
修改 Mandelbrot.cpp 中的代码，使用两个核来并行 Mandelbrot 生成。特别地，在线程 0 中计算图像的上半部分，在线程 1 中计算图像的下半部分。这种类型的问题分解被称为空间分解，因为图像的不同空间区域是由不同的处理器计算的。

2. Extend your code to utilize T threads for T ∈ {2, 4, 8, 16}, partitioning the image generation work into the appropriate number of horizontal blocks. You will need to modify the code in function workerThreadStart, to partition the work over the threads.
扩展代码以利用 t 线程来实现 t ∈{2,4,8,16}，将图像生成工作划分为适当数量的水平块。你需要修改 workerThreadStart 函数中的代码，在线程之间进行分区。

Note that the processor has 8 cores. Also, the active images have 599 rows (with another row added to detect array overrun), and so you must handle the case where the number of rows is not evenly divisible by the number of threads. In your write-up, produce a graph of speedup compared to the reference sequential implementation as a function of the

number of cores used for views 0, 1, and 2. Is speedup linear in the number of cores used? In your writeup hypothesize why this is (or is not) the case?
注意处理器有 8 个核。此外，活动图像有 599 行(添加了另一行以检测数组溢出)，因此必须处理行数不能被线程数整除的情况。在您的写作中，生成一个与参考顺序实现相比的加速度图，作为视图 0、1 和 2 使用的核数的函数。加速是否与使用的核心数量成线性关系？在你的文章中假设为什么会这样(或者不是)？

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

为了证实(或反驳)您的假设，通过在 workerThreadStart ()的开头和结尾插入计时代码来测量每个线程完成其工作所需的时间。你的测量结果如何解释你之前创建的加速图？

4. Modify the mapping of work to threads to improve speedup to 8× on view 0 and almost 8× on views 1 and 2 (if you're close to 8× that's fine, don't sweat it). You may not use any synchronization between threads. We expect you to come up with a single work decomposition policy that will work well for all thread counts; hard coding a solution specific to each configuration is not allowed! (Hint: There is a very simple static assignment that will achieve this goal, and no communication/synchronization among threads is necessary.) In your writeup, describe your approach and report the final 16-thread

修改工作到线程的映射，在视图 0 上将速度提高到 8 倍，在视图 1 和视图 2 上几乎提高到 8 倍(如果接近 8 倍就可以了，不用担心)。你不能在线程之间使用任何同步。我们希望您能够提出一个适用于所有线程计数的单一工作分解策略；不允许硬编码特定于每个配置的解决方案！(提示：有一个非常简单的静态分配来实现这个目标，线程之间不需要通信/同步在你的写作中，描述你的方法并报告最终的 16 线程

speedup obtained. Also comment on the difference in scaling behavior from 4 to 8 threads versus 8 to 16 threads.
还可以评论从 4 到 8 个线程与从 8 到 16 个线程的缩放行为的差异。

As a bonus (to you, not extra credit), once you have your threaded code running properly, you can run our interactive visualization tool of the Mandelbrot set. You will find this quite addictive. The program is implemented as the file mviz.py in the main assignment directory. Invoke it using, as command-line arguments, the command line you give to run the mandelbrot program. For example, you might give the command
作为额外的奖励(对您来说，不是额外的学分)，一旦您的线程代码正常运行，您就可以运行我们的交互式可视化工具 Mandelbrot 集。你会发现这很容易上瘾。该程序实现为主分配目录中的文件 mviz.py。调用它，作为命令行参数，使用你给的命令行来运行 mandelbrot 程序。例如，你可以给出命令

```
linux> ../mviz.py ./mandelbrot -t 16
/mviz.py./mandelbrot-t 16
```

When the program starts, it will display the list of single-character keyboard commands you can use to zoom and pan around the set. You will notice the fractal self similarity property, where common patterns keep occurring as you zoom deeper. You will also find that speedup you get from threading can greatly improve the interactive response.
当程序启动时，它会显示单字符键盘命令的列表，你可以使用它来缩放和平移设置。你会注意到分形自相似属性，当你缩放更深的时候，常见的模式不断出现。你还会发现线程处理的加速可以极大地提高交互式响应。

What you need to turn in:
你需要上交的内容:

1. Your report should contain the graphs, analyses, and answers specified above.
你的报告应该包含上面指定的图表、分析和答案。

2. Your report should describe the decomposition strategy you used to maximize speedup.
你的报告应该描述你用来最大化加速的分解策略。

3. The archive file you submit will contain your version of the file mandelbrot.cpp. This file should contain the best performing code you created. Any modifications you made should follow good coding conventions, in terms of indenting, variable names, and comments.
你提交的存档文件将包含你的 mandelbrot.cpp 文件版本。这个文件应该包含你创建的性能最好的代码。你所做的任何修改都应该遵循良好的编码约定，包括缩进、变量名和注释。

2 Problem 2: Vectorizing Code Using SIMD Intrinsics (20 points)
问题 2: 使用 SIMD Intrinsics 对代码进行矢量化(20 点)

Take a look at the function clampedExpSerial() in prob2_vecintrin/functions.cpp of the Assignment 1 code base. The function raises values[i] to the integer power given by exponents[i] for all elements of the input array and clamps the resulting values at 4.18. The function computes x^p based on the technique known

as exponentiation by squaring. Whereas the usual technique of multiplying together p copies of x requires $p - 1$ multiplications, iterative squaring requires at most $2 \log_2 p$ multiplications. For $p = 1000$, exponentiation by squaring requires less than 20 multiplications rather than 999. In Problem 2, your job is to vectorize this piece of code so that it can be run on a machine with SIMD vector instructions.
看看 Assignment 1 代码库的 prob2 _ vecintrin/functions.cpp 中的 clampedExpSerial ()函数。该函数将输入数组的所有元素的值[i]提升为指数[i]给出的整数幂，并将得到的值固定在 4.18。该函数基于称为平方幂的技术计算 xp。通常的 x 的 p 拷贝相乘需要 p-1 的乘法，而迭代平方最多需要 2log2p 的乘法。对于 p = 1000，平方乘法需要少于 20 次的乘法，而不是 999 次。在问题 2 中，你的工作是向量化这段代码，这样它就可以在一台带有 SIMD 向量指令的机器上运行。

We won’t ask you to craft an implementation using the SSE or AVX vector instrinsics that map to real vector instructions on moderns CPUs. Instead, to make things a little easier, we’re asking you to implement your version using 15-418’s “fake vector instrinics” defined in CMU418intrin.h. The CMU418intrin library provides you with a set of vector instructions that operate on vector values and/or vector masks. (These functions don’t translate to real vector instructions; instead we simulate these operations for you in our library, and provide feedback that makes for easier debugging.) As an example of using the 15-418 intrinsics, a vectorized version of the abs() function is given in functions.cpp. This example contains some basic vector loads and stores and manipulates mask registers. Note that the abs() example is only a simple example, and in fact the code does not correctly handle all inputs. (We will let you figure
我们不会要求你制作一个使用 SSE 或 AVX 向量 instinics 映射到现代 cpu 上真正的向量指令的实现。相反，为了让事情变得简单一点，我们要求您使用 CMU418intrin.h 中定义的 15-418 的“伪矢量 instrinics”来实现您的版本。CMU418intrin 库为您提供了一组矢量指令，可以对矢量值和/或矢量掩码进行操作。(这些函数不会转换成真正的矢量指令；相反，我们在我们的库中为您模拟这些操作，并提供反馈，使调试更加容易。)作为使用 15-418 intrinsic 的一个例子，在 functions.cpp 中给出了 abs ()函数的向量化版本。这个例子包含了一些基本的矢量加载，存储和操作掩码寄存器。注意 abs ()示例只是一个简单的示例，实际上代码并不能正确处理所有的输入。(我们会让你想想

out why.) You may wish to read through the comments and function definitions in CMU418intrin.h to know what operations are available to you.
为什么。)您可能希望通读 CMU418intrin.h 中的注释和函数定义，以了解您可以使用哪些操作。

Here are a few hints to help you in your implementation:
下面是一些帮助您实现的提示:

1. Every vector instruction is subject to an optional mask parameter. The mask parameter defines which lanes have their output “masked” for this operation. A 0 in the mask indicates a lane is masked, and so its value will not be overwritten by the results of the vector operation. If no mask is specified in the operation, no lanes are masked. (This is equivalent to providing a mask of all ones.) Your solution will need to use multiple mask registers and various mask operations provided in the library.

每条向量指令都有一个可选的掩码参数。Mask 参数定义了哪些车道的输出为这个操作“屏蔽”。掩码中的 0 表示一条车道被屏蔽了，因此它的值不会被向量运算的结果覆盖。如果在操作中没有指定掩码，则没有车道被掩码。(这相当于提供了所有的一个掩码你的解决方案需要使用多个掩码寄存器和库中提供的各种掩码操作。

2. You will find the _cmu418_cntbits() function to be helpful in this problem.

您会发现 _ cmu418 _ cntbits ()函数对这个问题很有帮助。

3. You must handle the case where the total number of loop iterations is not a multiple of SIMD vector width. We suggest you test your code with ./vrun -s 3. You might find _cmu418_init_ones() helpful.

你必须处理循环迭代的总次数不是 SIMD 向量宽度的倍数的情况。我们建议您使用。/vrun-s 3.你可能会发现 _cmu418 _ init _ ones ()很有帮助。

4. Use command-line option -l to print a log of executed vector instruction at the end. Insert calls to function addUserLog() in your vector code to add customized debugging information to the log.

使用命令行选项 -l 在最后打印执行的矢量指令的日志。Insert 调用函数 addUserLog ()在矢量代码中添加自定义的调试信息到日志中。

The output of the program will tell you if your implementation generates correct output. If there are incorrect results, the program will print the first one it finds and print out a table of function inputs and outputs.[1] Your function’s output is after “output = ”, which should match with the results after “gold = ”. The program also prints out a list of statistics describing utilization of the 15418 fake vector units. You should consider the performance of your implementation to be the value “Total Vector Instructions”. “Vector Utilization” shows the percentage of vector lanes that are enabled.
程序的输出会告诉你你的实现是否生成了正确的输出。如果有不正确的结果，程序将打印它找到的第一个结果，并打印出函数输入和输出的表格。1 你的函数的输出在“ output =”之后，应该与“ gold =”之后的结果相匹配。程序还会打印出描述 15418 假向量单位使用情况的统计数据列表。你应该考虑你的实现的性能是值“ Total Vector Instructions”。矢量利用率(Vector Utilization)显示了启用的矢量车道的百分比。

What you need to do:
你需要做的:

1. Implement a vectorized version of clampedExpSerial() as the function clampedExpVector() in file functions.cpp. Your implementation should work with any combination of input array size N and vector width W .

在 functions.cpp 文件中实现 clampedExpSerial ()的矢量化函数 clampedExpVector ()。你的实现应该与任何输入数组大小 n 和向量宽度 w 的组合一起工作。

2. Run ./vrun -s 10000 and sweep the vector width over the values {2, 4, 8, 16, 32}. Record the resulting vector utilization. You can do this by changing the defined value of VECTOR_WIDTH in CMU418intrin.h and recompiling the code each time. 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 .

快跑。/vrun-s 10000 并将矢量宽度扫描到值{2,4,8,16,32}上。记录最终的矢量利用率。你可以通过修改 CMU418intrin.h 中定义的 VECTOR _ width 的值并每次重新编译代码来做到这一点。当 w 变化时，向量利用率变化了多少？解释这些变化的原因以及利用率对矢量宽度的敏感程度。解释矢量指令的总数如何随 w 而变化。

3. Extra credit: (1 point) Implement a vectorized version of arraySumSerial() as the function arraySumVector() in file functions.cpp. Your implementation may assume that W is a factor of the input array size N. Whereas the serial implementation has O(N) span, your implemen-tation should have at most $O(N/W + \log_2 W)$ span. You may find the hadd and interleave operations useful.

加分: (1 分)在 functions.cpp 文件中实现一个矢量化的 arraySumSerial ()作为函数 arraySumVector ()。您的实现可以假设 w 是输入数组大小 n 的一个因子，而串行实现具有 o (n)跨度，而您的实现最多应该具有 o (n/w + log2w)跨度。你可能会发现 hadd 和交织操作很有用。

[1]If it finds a mismatch in row 599, that indicates that your program overran the image array.
如果在第 599 行中发现不匹配，则表示程序溢出了图像数组。

What you need to turn in:
你需要上交的内容:

1. Your report should contains tables giving the vector utilizations and total vector instructions for the different values of W .
你的报告应该包含一些表格，这些表格给出了 w 的不同值的矢量利用率和总矢量指令。

2. Your report should contain the analyses and answers to the questions listed above.
你的报告应该包含上述问题的分析和答案。

3. If you did the extra credit problem, state in the report whether or not your code passed the correctness test.
如果你做了额外的学分问题，在报告中说明你的代码是否通过了正确性测试。

4. The archive file you submit will contain your version of the file functions.cpp. This file should contain the best performing code you created. Any modifications you made should follow good coding conventions, in terms of indenting, variable names, and comments.
你提交的存档文件将包含你的文件 functions.cpp 版本。这个文件应该包含你创建的性能最好的代码。你所做的任何修改都应该遵循良好的编码约定，包括缩进、变量名和注释。

3 Problem 3: Using Instruction-level Parallelism in Fractal Generation (20 points)

问题 3: 在分形生成中使用指令层级平行(20 点)

Now that you have seen how SIMD execution can operate on multiple data values simultaneously, we will explore how these principles can speed up a conventional C++ program. In this case, we will exploit the parallel execution capability provided in the multiple, pipelined functional units of an out-of-order processor. This exercise will give us the chance to analyze how well a program makes use of these units and how this limits program performance.
既然您已经了解了 SIMD 执行是如何同时操作多个数据值的，那么我们将探讨这些原则是如何加速传统 c + + 程序的。在这种情况下，我们将利用在无序处理器的多个流水线功能单元中提供的并行执行能力。这个练习将使我们有机会分析一个程序如何使用这些单元，以及这如何限制程序性能。

You will want to refer to material in Computer Systems: A Programmer’s Perspective, third edition (CS:APP3e) that you perhaps did not study carefully in whatever version of 15-213 you took. In particular, you will find Sections 3.11 (floating-point code) and Sections 5.7–5.10 (out-of-order execution and instruction-level par-allelism) to be helpful. Copies of the book are on reserve in the Sorrell’s Library. Don’t try to get by with an earlier edition of the book.
您可能想参考《计算机系统: 程序员的视角》第三版(CS: APP3e)中的材料，这些材料在您选择的 15-213 的任何版本中可能都没有仔细研究过。特别是，您会发现第 3.11 节(浮点代码)和第 5.7-5.10 节(乱序执行和指令级并行)非常有用。这本书的副本保留在 Sorrell’s Library。不要试图用这本书的早期版本来应付。

All of the code for this problem is provided in the directory prob3 mandelbrot ilp. Your job will be to measure performance and understand it. Compile and run the code on view 0. This is a

useful benchmark, since all of the iterations hit the maximum limit of 256, yielding a minimum of overhead. You will find that the reference implementation, as given by the function mandel ref requires around 13 clock cycles per iteration.
这个问题的所有代码都在 prob3 mandelbrot ilp 目录中提供。你的工作将是测量性能并理解它。编译并在视图 0 上运行代码。这是一个有用的基准，因为所有的迭代都达到了 256 的最大限度，产生了最小的开销。你会发现函数 mandel ref 给出的参考实现每次迭代大约需要 13 个时钟周期。

Figure 1 shows the assembly code generated for the inner loop of mandel ref, with some comments added to help you understand how the integer and floating-point registers are used. Add more annotation to this code to describe how it relates to the original C++ code. (A copy of the code is provided in the file

mandel ref loop.s.) Explain: How is the expression 2.f * z re * z im is implemented?
图 1 显示了为 mandel ref 的内部循环生成的汇编代码，并添加了一些注释，以帮助您理解如何使用整数和浮点寄存器。在这段代码中添加更多的注释来描述它与原始 c + + 代码的关系。(代码的副本在文件 mandel ref loop.s 中提供)解释: 表达式 2 怎么样。F * z re * z im 是如何实现的？

Create a data-flow diagram indicating the data dependencies formed by successive iterations of the loop, similar to Figure 5.14(b) of CS:APP3e. You need not show the integer operations, since these do not affect the program performance.
创建一个数据流图，指示循环的连续迭代所形成的数据依赖关系，类似于 CS: APP3e 的图 5.14(b)。你不需要显示整数操作，因为它们不会影响程序的性能。

The Gates-cluster machines are based on what Intel labels its “Coffee Lake” microarchitecture. This is very similar to Haswell microarchitecture described in Section 5.7.2 of CS:APP3e, but with slightly more and faster functional units. You can find out more about the microarchitecture in
盖茨集群机器是基于英特尔所称的“咖啡湖”微架构。这与 CS: APP3e 第 5.7.2 节中描述的 Haswell 微架构非常相似，但是功能单元更多、更快。你可以在这里找到更多关于微架构的信息

```
#  This is the inner loop of mandel_ref
这是 mandel _ ref 的内部循环
#  Parameters are passed to the function as follows:
参数如下所示传递给函数:
#      %xmm0: c_re
% xmm0: c _ re
#      %xmm1: c_im
% xmm1: c _ im
#      %edi:  count
% edi: 计数
#  Before entering the loop, the function sets registers
在进入循环之前，函数设置寄存器
#  to initialize local variables:
初始化局部变量:
#      %xmm2: z_re = c_re
% xmm2: z _ re = c _ re
#      %xmm3: z_im = c_im
% xmm3: z _ im = c _ im
#      %eax:  i = 0
% eax: i = 0

.L123:
L123:
        vmulss      %xmm2, %xmm2, %xmm4
        Vmulss% xmm2,% xmm2,% xmm4
        vmulss      %xmm3, %xmm3, %xmm5
        Vmulss% xmm3,% xmm3,% xmm5
        vaddss      %xmm5, %xmm4, %xmm6
        Vaddss% xmm5,% xmm4,% xmm6
        vucomiss            .LC0(%rip), %xmm6       # Compare to 4.f
        LC0(% rip) ,% xmm6 # 与 4. f 比较
        ja          .L126
        是的 L126
        vaddss      %xmm2, %xmm2, %xmm2
        Vaddss% xmm2,% xmm2,% xmm2
        addl        $1, %eax
        1 美元，税率%
        cmpl        %edi, %eax        # Set condition codes for jne below
        Cmpl% edi,% eax # Set condition codes for jne below
        vmulss      %xmm3, %xmm2, %xmm3
        Vmulss% xmm3,% xmm2,% xmm3
        vsubss      %xmm5, %xmm4, %xmm2
        Vsubss% xmm5,% xmm4,% xmm2
```

```
vaddss      %xmm3, %xmm1, %xmm3
Vaddss% xmm3,% xmm1,% xmm3
vaddss      %xmm2, %xmm0, %xmm2
Vaddss% xmm2,% xmm0,% xmm2
jne         .L123
Jne. L123
```

Figure 1: Assembly code for inner loop of function mandel ref
图 1: 函数 mandel ref 内部循环的汇编代码

en.wikichip.org/wiki/intel/microarchitectures/coffee lake
En.wikichip. org/wiki/intel/microarchitectures/coffee lake

Coffee Lake shares the same CPU core design as Intel's previous two server processors, called Kaby Lake and Skylake. You can read about the details of the core design here (search for “Skylake”):
Coffee Lake 与 Intel 之前的两个服务器处理器(分别叫做 Kaby Lake 和 Skylake)有着相同的 CPU 核心设计。你可以在这里阅读核心设计的细节(搜索“ Skylake”) :

www.agner.org/optimize/microarchitecture.pdf
Www.agner. org/optimize/microarchitecture. pdf

Pages 151–152 of this document describe the functional units (the wikichip.org link has diagrams). In particular, the processor has two units for floating-point arithmetic. These are fully pipelined, able to start new operations on every clock cycle. For floating point on xmm registers, unit 0 can perform addition, multiplication, and division, while unit 1 can only perform addition and multiplication. Unit 5 can do floating-point addition, but only using legacy x87 floating-point instructions. Both operations have a latency of 4 clock cycles (except for addition in unit 5, which takes 3 cycles). A separate functional unit can perform the floating-point comparison required by the vucomiss instruction.
本文档的第 151-152 页描述了功能单元(wikichip.org 链接有图表)。特别是，处理器有两个浮点运算单元。它们是完全流水线的，能够在每个时钟周期启动新的操作。对于 xmm 寄存器上的浮点数，单元 0 可以执行加法、乘法和除法，而单元 1 只能执行加法和乘法。Unit 5 可以做浮点加法，但是只能使用传统的 x87 浮点指令。这两个操作都有 4 个时钟周期的延迟(除了第 5 单元的加法，它需要 3 个周期)。一个单独的功能单元可以执行 vucomiss 指令所需的浮点比较。

Based on these latencies and the data-flow diagram you have created, determine the latency bound for function mandel ref. How close does the measured performance come to reaching this bound?
根据这些延迟和您已经创建的数据流图，确定函数 mandel ref 的延迟界限。测量的性能距离这个界限有多近？

It is difficult to see how to speed up the individual iterations through increased instruction-level parallelism. However, it is possible to increase the level of activity in the loop by processing multiple candidate points. This will enable increasing the throughput of the processing to yield better performance than is dictated by the latency bound.
很难看出如何通过增加指令层级平行性来加快单个迭代的速度。然而，通过处理多个候选点来增加循环中的活动级别是可能的。这将能够增加处理的吞吐量，从而产生比延迟限制更好的性能。

The file mandelbrot.cpp contains a macro MANDEL BODY and its instantiations to generate the functions mandel parU for U ranging from 1 to 8. The idea is to process U points in parallel, using a style of programming similar to SIMD execution. We represent the multiple data with arrays of size U, and write conventional code that uses looping to process all U elements. As with SIMD, the loop exits only when all points have completed their iterations, using a bit vector to determine which elements should be updated. The compiler automatically unrolls all of these loops, and so the generated code is equivalent to what would be obtained by explicitly writing the computation for all U elements. (You can see the generated code in the file mandelbrot.s, generated when you compiled the code for this problem.) The function mandelbrotParallel exploits this capability by computing U adjacent rows of the array in parallel.
Cpp 文件 mandelbrot.cpp 包含一个宏 MANDEL BODY 及其实例化，用于生成范围为 1 到 8 的函数 MANDEL parU。其思想是并行处理 u 点，使用类似于 SIMD 执行的编程风格。我们用 u 大小的数组来表

示多个数据，然后编写传统的代码，使用循环来处理所有 u 元素。与 SIMD 一样，循环只有在所有点完成迭代后才会退出，使用一个位向量来确定哪些元素应该被更新。编译器会自动展开所有这些循环，因此生成的代码相当于通过显式编写所有 u 元素的计算得到的代码。(你可以在 mandelbrot.s 文件中看到生成的代码，它是在编译这个问题的代码时生成的 mandelbrotParallel 函数通过并行计算数组中相邻的 u 行来利用这种能力。

Running the program on view 0, you will see that it reaches a peak performance of around 5.1 clock cycles per iteration for U = 5. (That is, each execution of the loop requires around 25.5 cycles, but it performs an iteration for five points.) That’s an improvement over the original performance, but perhaps not quite as significant as one may hope.
在视图 0 上运行这个程序，你会发现当 u = 5 时，它的最高性能是每次迭代 5.1 个时钟周期。(也就是说，循环的每次执行大约需要 25.5 个周期，但是它执行了 5 个点的迭代这是对原始性能的一个改进，但可能没有人们希望的那么显著。

Considering the floating-point operations, what is the highest throughput bound imposed by the functional units? You should consider the bounds imposed by multiplication, addition, and the two in combination. How close does the measured performance come to reaching this bound?
考虑到浮点操作，功能单元强加的最高吞吐量是多少？你应该考虑乘法、加法和两者的组合所强加的界限。测量的性能离达到这个界限有多远？

If you could modify the generated assembly code, do you think you could lower the throughput bound? Extra Credit: (1 point) Modify the C++ code for both the reference and the parallel versions to (slightly) improve the performance of the parallel version, while still passing the functionality tests. You may alter the computed function, but only in a way that demonstrates how a better compiler could generate faster code.
如果你可以修改生成的汇编代码，你认为你可以降低吞吐量上限吗？额外信用: (1 点)修改 c + + 代码的参考和并行版本，以(略微)提高性能的并行版本，同时仍然通过了功能测试。你可以修改计算函数，但只能通过演示更好的编译器如何生成更快的代码来实现。

As a bonus, you can run the interactive visualization tool mviz.py of the ILP-enhanced version the Man-delbrot computation with U = 5. Run with the command line:
作为奖励，您可以运行交互式可视化工具 mviz.py 的 ilp 增强版 Man-delbrot 计算，u = 5。使用命令行运行:

```
linux> ../mviz.py ./mandelbrot
Linux > . ./mviz.py./mandelbrot
```

What you need to turn in:
你需要上交的内容:

1. An annotated version of the assembly code for the main loop of mandel ref (the annotations should be made in the file mandel ref loop.s so that they will be submitted to Autolab, and you also need to show the content of annotated version in your report.pdf).
Mandel ref 主循环的汇编代码的注释版本(注释应该在 mandel ref loop.s 文件中进行，以便提交给 Autolab，您还需要在 report.pdf 中显示注释版本的内容)。

2. Data-flow diagram showing the loop-carried dependencies among the floating-point values.
数据流图，显示浮点值之间的循环依赖关系。

3. An analysis of the latency bound of mandel ref and how the measured performance compares.
分析 mandel ref 的延迟界限以及如何比较测量的性能。

4. An analysis of the throughput bound for the parallel versions, and how the measured performance compares.
分析并行版本的吞吐量限制，以及测量的性能如何比较。

5. Ideas for how a compiler could generate code that runs faster on this particular microarchitecture.
关于编译器如何在这个特殊的微架构上生成运行更快的代码的想法。

6. (Extra credit.) A demonstration of how the compiler could generate code that runs faster.
演示编译器如何生成运行更快的代码。

4 Problem 4: Parallel Fractal Generation Using ISPC (20 points)
问题 4: 使用 ISPC 生成并行分形(20 分)

The code for this problem is in the subdirectory prob4_mandelbrot_ispc. Now that you're comfort-able with SIMD execution, we'll return to parallel Mandelbrot fractal generation. As in Problem 1, Problem 4 computes a Mandelbrot fractal image, but it achieves even greater speedups by utilizing the SIMD execu-tion units within each of the 8 cores. We will also see that the ILP-enhancement techniques used in Problem 3 can, in principle, be applied to ISPC code.
这个问题的代码在子目录 prob4 _ mandelbrot _ ispc 中。既然你对 SIMD 的执行很满意，我们就回到并行的 Mandelbrot 分形生成。与问题 1 一样，问题 4 计算 Mandelbrot 分形图像，但是它通过利用 8 个核中每个核中的 SIMD 执行单元来实现更大的加速。我们还将看到，在问题 3 中使用的 ilp 增强技术原则上可以应用于 ISPC 代码。

In Problem 1, you parallelized image generation by creating one thread for each processing core in the system. Then, you assigned parts of the computation to each of these concurrently executing threads. Instead of specifying a specific mapping of computations to concurrently executing threads, Problem 4 uses ISPC language constructs to describe independent computations. These computations may be executed in parallel without violating program correctness. In the case of the Mandelbrot image, computing the value of each pixel is an independent computation. With this

information, the ISPC compiler and runtime system take on the responsibility of generating a program that utilizes the CPUs collection of parallel execution resources as efficiently as possible.
在问题 1 中，你通过为系统中的每个处理核创建一个线程来并行化图像生成。然后，你将计算的一部分分配给这些并发执行的线程。问题 4 使用 ISPC 语言构造来描述独立的计算，而不是指定一个特定的计算映射到并发执行的线程。这些计算可以在不违反程序正确性的情况下并行执行。在 Mandelbrot 图像的情况下，计算每个像素的值是一个独立的计算。有了这些信息，ISPC 编译器和运行时系统负责生成一个程序，该程序尽可能高效地利用并行执行资源的 cpu 集合。

4.1 Problem 4, Part 1. A Few ISPC Basics (5 of 20 points)
4.1 问题 4，第 1 部分。一些 ISPC 基础知识(20 分中的 5 分)

When reading ISPC code, you must keep in mind that, although the code appears much like C/C++ code, the ISPC execution model differs from that of standard C/C++. In contrast to C, multiple program instances of an ISPC program are always executed in parallel on the CPU's SIMD execution units. The number of program instances executed simultaneously is determined by the compiler (and chosen specifically for the underlying machine). This number of concurrent instances is available to the ISPC programmer via the built-in variable programCount. ISPC code can reference its own program instance identifier via the built-in programIndex. Thus, a call from C code to an ISPC function can be thought of as spawning a
在阅读 ISPC 代码时，必须记住，尽管该代码看起来很像 c/c + + 代码，但 ISPC 的执行模型与标准 c/c + + 的执行模型不同。与 c 不同，一个 ISPC 程序的多个程序实例总是在 CPU 的 SIMD 执行单元上并行执行。同时执行的程序实例的数量由编译器决定(并专门为底层机器选择)。ISPC 程序员可以通过内置变量 programCount 获得并发实例的数量。ISPC 代码可以通过内置 programIndex 引用自己的程序实例标识符。因此，从 c 代码到 ISPC 函数的调用可以被认为产生了一个

group of concurrent ISPC program instances (referred to in the ISPC documentation as a gang). The gang of instances runs to completion, then control returns back to the calling C code.
一组并发的 ISPC 程序实例(在 ISPC 文档中称为一组)。一组实例运行到完成，然后控制返回到调用 c 代码。

As an example, the following program uses a combination of regular C code and ISPC code to add two 1024-element vectors. As discussed in class, since each instance in a gang is independent and performs the exact same program logic, execution can be accelerated via SIMD instructions.
作为一个例子，下面的程序使用常规 c 代码和 ISPC 代码的组合来添加两个 1024 元素向量。正如我们在课堂上讨论的那样，由于一组中的每个实例都是独立的，并且执行完全相同的程序逻辑，因此可以通过 SIMD 指令来加速执行。

A simple ISPC program is given below. First, the C program, which calls the ISPC-generated code:
下面给出了一个简单的 ISPC 程序。首先，c 程序调用 ISPC 生成的代码:

```
------------------------------------------------------------------------
------------------------------------------------------------------------
C program code: myprogram.cpp
C 程序代码: myprogram.cpp
------------------------------------------------------------------------
------------------------------------------------------------------------
const int TOTAL_VALUES = 1024;
Const int TOTAL _ values = 1024;
float a[TOTAL_VALUES];
浮动 a [ TOTAL _ values ] ;
float b[TOTAL_VALUES];
浮点数 b [ TOTAL _ values ] ;
float c[TOTAL_VALUES]
浮点 c [ TOTAL _ values ]

//    Initialize arrays a and b here.
在这里初始化数组 a 和 b。
. . .
. . .

sum(TOTAL_VALUES, a, b, c);
和(TOTAL _ values，a，b，c) ;

// Upon return from sumArrays, result of a + b is stored in c.
//从 sumarray 返回时，a + b 的结果存储在 c 中。
```

The function sum() called by the C code is generated by compiling the following ISPC code:
C 代码调用的函数 sum ()是通过编译以下 ISPC 代码生成的:

```
------------------------------------------------------------------------
------------------------------------------------------------------------
ISPC code: myprogram.ispc
ISPC 代码: myprogram.ISPC
------------------------------------------------------------------------
------------------------------------------------------------------------

export sum(uniform int N, uniform float* a, uniform float* b, uniform float* c) {
```

```
Export sum (uniform int n, uniform float * a, uniform float * b, uniform float * c){
        //    Assumes programCount divides N evenly. for (int
        i=0; i<N; i+=programCount)
        假设 programCount 均匀地除以 n。 for (int i = 0; i < n; i
        + = programCount)
        {
                c[programIndex + i] = a[programIndex + i] + b[programIndex + i];
                C [ programIndex + i ] = a [ programIndex + i ] + b [ programIndex + i ] ;
        }
}
```

The ISPC program code above interleaves the processing of array elements among program instances. Note the similarity to Problem 1, where you statically assigned parts of the image to threads.

上面的 ISPC 程序代码在程序实例之间交织数组元素的处理。请注意与问题 1 的相似之处，在问题 1 中，你静态地将图像的一部分分配给线程。

However, rather than thinking about how to divide work among program instances (that is, how work is mapped to execution units), it is often more convenient, and more powerful, to instead focus only on the partitioning of a problem into independent parts. ISPCs foreach construct provides a mechanism to express problem decomposition. Below, the foreach loop in the ISPC function sum2() defines an iteration space where all iterations are independent and therefore can be carried out in any order. ISPC handles the assignment of loop iterations to concurrent program instances. The difference between sum() and sum2() below is subtle, but very important. sum() is imperative: it describes how to map work to concurrent instances. The sum2() function below is declarative: it specifies only the set of work to be performed.

然而，与其考虑如何在程序实例之间分配工作(即如何将工作映射到执行单元)，通常只关注于将问题划分为独立的部分更为方便，也更为强大。ISPCs foreach 构造提供了一种表达问题分解的机制。下面，ISPC 函数 sum2()中的 foreach 循环定义了一个迭代空间，其中所有迭代都是独立的，因此可以按照任何顺序执行。ISPC 处理并发程序实例的循环迭代分配。下面的 sum ()和 sum2()之间的区别很微妙，但是非常重要。Sum ()是必要的: 它描述了如何将工作映射到并发实例。下面的 sum2()函数是声明性的: 它只指定要执行的工作集。

10

ISPC code:
ISPC 代码:


```
export sum2(uniform int N, uniform float* a, uniform float* b, uniform float* c) {
Export sum2(uniform int n， uniform float * a， uniform float * b， uniform float * c){
      foreach (i = 0 ... N)
      Foreach (i = 0... n)
      {
            c[i] = a[i] + b[i];
            C [ i ] = a [ i ] + b [ i ] ;
      }
}
```

Before proceeding, you are encouraged to familiarize yourself with ISPC language constructs by reading through the ISPC walkthrough available at http://ispc.github.io/example.html. The example program in the walkthrough is almost exactly the same as Problem 4's implementation of mandelbrot_ispc() in mandelbrot.ispc. In the assignment code, we have changed the bounds of the foreach loop to yield a more straightforward implementation.
在继续之前，鼓励您通过阅读 http://ISPC.github.io/example. html 上提供的 ISPC 演练来熟悉 ISPC 语言构造。演练中的示例程序几乎与问题 4 在 mandelbrot.ispc 中 mandelbrot _ ispc ()的实现完全相同。在赋值代码中，我们已经改变了 foreach 循环的界限，以产生一个更直接的实现。

What you need to do:
你需要做的:

1. Compile and run the program mandelbrot.ispc. The ISPC compiler is configured to emit 8-wide AVX vector instructions. 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? Hint: Consider the characteristics of the computation you are performing. What parts of the image present challenges for SIMD execution? Comparing the performance of rendering the different views of the Mandelbrot set may help confirm your hypothesis.
编译并运行 mandelbrot.ispc 程序。ISPC 编译器被配置为发出 8 个 AVX 向量指令。根据你对这些 cpu 的了解，你期望的最大加速度是多少？为什么你观察到的数字会低于这个理想值？提示: 考虑你正在执行的计算的特点。图像的哪些部分对 SIMD 的执行提出了挑战？比较渲染 Mandelbrot 集合不同视图的性能可能有助于证实你的假设。

   We remind you that for the code described in this subsection, the ISPC compiler maps gangs of program instances to SIMD instructions executed on a single core. This parallelization scheme differs from that of Problem 1, where speedup was achieved by running threads on multiple cores.
   我们提醒你，对于本小节中描述的代码，ISPC 编译器将程序实例组映射到在单个核上执行的 SIMD 指令。这种并行化方案不同于问题 1，在问题 1 中，加速是通过在多个核上运行线程来实现的。

4.2 Problem 4, Part 2: Combining instruction-level and SIMD parallelism (7 of 20 points)
4.2 问题 4，第 2 部分: 结合指令级和 SIMD 并行性(20 分中的 7 分)

The floating-point functional units in the Gates cluster processors are fully vectorized. They can perform 8-wide additions and multiplications on single-precision data. Thus, the same techniques we used to exploit instruction-level parallelism can be combined with the SIMD execution targeted by ISPC.
Gates 集群处理器中的浮点功能单元是完全向量化的。它们可以在单精度数据上执行 8 个宽度的加法和乘法。因此，我们用于开发指令层级平行性的相同技术可以与 ISPC 所针对的 SIMD 执行相结合。

The function mandel par2 provides two-way parallelism of the Mandelbrot computation. (Unfortunately, the ISPC compiler cannot handle the small loops and arrays we used in Problem 3, and so this code must explicitly duplicate each line of code. It must also pass the point data as separate scalar values, rather than as arrays.) Your job is to modify the function mandelbrot ispc par2 to make use of this parallel form. As we did in Problem 3, your code should process two rows per pass. You should make use of the ISPC foreach construct, but modifying it to only do half as many passes. You also should handle the case where the height of the array is not a multiple of two.
函数模型 par2 提供了 Mandelbrot 计算的双向并行性。(不幸的是，ISPC 编译器无法处理我们在问题 3 中使用的小循环和数组，因此这段代码必须显式地重复每行代码。它还必须将点数据作为单独的标量值传递，而不是作为数组你的工作是修改函数 mandelbrot ispc par2 来使用这种并行形式。就像我们在问题 3 中做的那样，你的代码应该每次处理两行代码。你应该使用 ISPC foreach 构造，但是修改它只需要执行一半的遍历。你还应该处理数组高度不是 2 的倍数的情况。

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?
这种双向并行比普通的 ISPC 版本提高了多少速度？在不同的输入(例如，不同的视图)之间是否存在差异？什么时候值得付出努力？

11

4.3 Problem 4, Part 3: ISPC Tasks (8 of 20 points)
4.3 问题 4，第三部分: ISPC 任务(20 分中的 8 分)

ISPC’s SPMD execution model and the foreach mechanism facilitate the creation of programs that utilize SIMD processing. The language also provides the launch mechanism to utilize multiple cores in an ISPC computation, via a lightweight form of threading known as tasks.
ISPC 的 SPMD 执行模型和 foreach 机制有助于创建利用 SIMD 处理的程序。该语言还提供了一种启动机制，通过一种称为任务的轻量级线程形式，在 ISPC 计算中利用多个核。

See the launch command in the function mandelbrot_ispc_withtasks() in the file mandelbrot.ispc. This command launches multiple tasks (always 4 in the starter code). Each task defines a computation that will be executed by a gang of ISPC program instances. As given by the function mandelbrot_ispc_task(), each task computes a region of the final image with dimensions BLOCK WIDTH × BLOCK HEIGHT. Simi-
参见函数 mandelbrot _ ispc _ withtasks ()文件 mandelbrot.ispc 中的启动命令。这个命令启动多个任务(在启动代码中总是 4 个)。每个任务定义一个由一组 ISPC 程序实例执行的计算。由函数 mandelbrot _ ispc _ task ()给出，每个任务计算最终图像的一个区域，其尺寸为 BLOCK WIDTH × BLOCK HEIGHT。Simi-
lar to how the foreach construct defines loop iterations that can be carried out in any order (and in parallel by ISPC program instances), the tasks created by this launch operation can be processed in any order (and in parallel on different CPU cores).
对于 foreach 构造如何定义循环迭代，循环迭代可以按任何顺序执行(并且由 ISPC 程序实例并行执行)，这个启动操作创建的任务可以按任何顺序(并且在不同的 CPU 核上并行执行)进行处理。

What you need to do:
你需要做的:

1. Run mandelbrot_ispc with the command-line option “--tasks.” What speedup do you observe on view 1? What is the speedup over the version of mandelbrot_ispc that does not partition that computation into tasks?
   使用命令行选项“—— tasks”运行 mandelbrot _ ispc 在视图 1 中你观察到了什么样的加速？什么是 mandelbrot _ ispc 版本的加速，没有将计算划分为任务？

2. The starter code always spawns 4 tasks. This works for the image size, BLOCK WIDTH, and BLOCK HEIGHT we have given you, but it will not work in general. Modify the calculation of taskCount before launching tasks to always work correctly, for any image and block size.
   初始代码总是产生 4 个任务。这个方法适用于我们给出的图像大小、块宽度和块高度，但是一般情况下不适用。在启动任务之前，修改任务计数的计算，使任何图像和块大小都能正常工作。

   Note: Your code must correctly handle the case where the block width or height does not evenly divide the image width or height.
   注意: 您的代码必须正确处理块宽度或高度不能均匀分割图像宽度或高度的情况。

3. There is a simple way to improve the performance of mandelbrot_ispc --tasks by increasing the number of tasks the code creates. By only changing values of BLOCK WIDTH, and BLOCK HEIGHT, you should be able to achieve performance that exceeds the sequential version of the code by about 20–22 times! What region sizes and shapes work best? (Do you prefer tall, wide, or square blocks?)

有一个简单的方法可以提高 mandelbrot _ ispc 的性能——通过增加代码创建的任务数来完成任务。通过只更改 BLOCK WIDTH 和 BLOCK HEIGHT 的值，您应该能够实现性能超过代码的顺序版本大约 20-22 倍！什么样的区域大小和形状效果最好？(你喜欢高的，宽的，还是方形的

4. Extra Credit: (1 point) What are differences between the Pthread abstraction (used in Problem 1) and the ISPC task abstraction? There are some obvious differences in semantics between the (create/join and (launch/sync) mechanisms, but the implications of these differences are more subtle. Here's a thought experiment to guide your answer: what happens when you launch 10,000 ISPC tasks? What happens when you launch 10,000 pthreads?

Extra Credit: (1 点) Pthread 抽象(在问题 1 中使用)和 ISPC 任务抽象之间有什么区别？(create/join 和(launch/sync)机制之间在语义上有一些明显的差异，但是这些差异的含义更加微妙。这里有一个思维实验来指导你的答案: 当你启动 10000 个 ISPC 任务时会发生什么？当你启动 10000 个 pthreads 时会发生什么？

Some of you may be thinking: "Hey wait! Why are there two different mechanisms (foreach and launch) for expressing independent, parallelizable work to the ISPC system? Couldn't the system just partition the many iterations of foreach across all cores and also emit the appropriate SIMD code for the cores?"
有些人可能会想: "嘿，等等！为什么有两种不同的机制(foreach 和 launch)来表达独立的、可并行的 ISPC 系统工作？难道系统不能将 foreach 的许多迭代划分到所有的核上，同时为核发出适当的 SIMD 代码吗?"

Great question! Glad you asked! There are a lot of possible answers; we'll talk more in lecture.
好问题! 很高兴你问了! 有很多可能的答案; 我们将在讲座中讨论更多。

As a bonus, you can run the interactive visualization tool mviz.py of the ISPC versions of the Mandelbrot computation. Run with either the command line:
作为奖励，你可以运行交互式可视化工具 mviz.py 的 ISPC 版本的 Mandelbrot 计算。使用以下命令行运行:

```
linux> ../mviz.py ./mandelbrot_ispc
Linux > ./mviz.py./mandelbrot _ ispc
```

or
或者

```
linux> ../mviz.py ./mandelbrot_ispc -t
Linux > ./mviz.py./mandelbrot _ ispc-t
```

You will see how the speedup improves interactivity.
您将看到加速是如何提高交互性的。

What you need to turn in:
你需要上交的内容:

1. Your report must contain answers to the questions listed above.
您的报告必须包含上述问题的答案。

2. Your report should describe performance gains you get from SIMD, SIMD+ILP, and threaded paral-lelism.
你的报告应该描述你从 SIMD、 SIMD + ilp 和线程并行处理中获得的性能提升。

3. Your answer to the extra credit question, if any.
你对额外学分问题的回答，如果有的话。

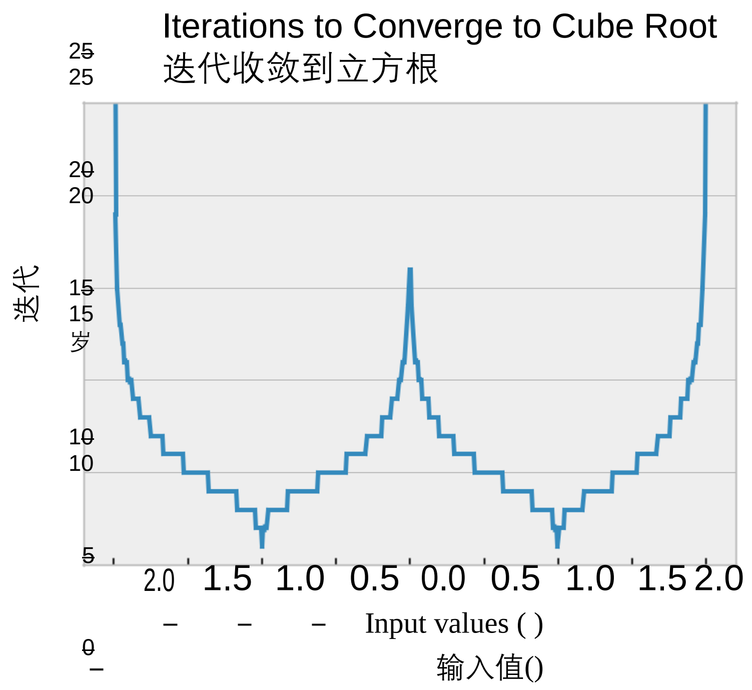
4. The archive file you submit will contain your version of the file mandelbrot.ispc. This file should contain the best performing code you created. Any modifications you made should follow good coding conventions, in terms of indenting, variable names, and comments.
你提交的存档文件将包含你的 mandelbrot.ispc 文件版本。这个文件应该包含你创建的性能最好的代码。你所做的任何修改都应该遵循良好的编码约定，包括缩进、变量名和注释。

5 Problem 5: Iterative Cubic Root (10 points)
问题 5: 迭代立方根(10 点)

The code for this problem is in the subdirectory prob5_cuberoot. Problem 5 concerns the program cuberoot, generated from an ISPC program that computes the cube root of 20 million random numbers between 0 and 3. For value s, it uses the iterative Newton's method (named after Isaac Newton) to solve the equation $1/x^3 = s^2$. This gives a value $x \approx s^{-2/3}$. Multiplying x by s gives an approximation to $s^{1/3}$.
这个问题的代码在 prob5 _ cuberroot 子目录中。问题 5 涉及 cuberoot 程序，由 ISPC 程序生成，该程序计算 0 到 3 之间 2000 万个随机数的立方根。对于值 s，它使用迭代牛顿法(以艾萨克牛顿命名)来求解方程 1/x3 = s2。这给出了一个值 x ≈ s-2/3。X 乘以 s 得到 s1/3 的近似值。

The value 1.0 is used as the initial guess in this implementation. The graph below shows the number of iterations required for the program to converge to an accurate solution for values in the open interval (−2, 2). (The implementation does not converge for inputs outside this range). Notice how the rate of convergence depends on how close the solution is to the initial guess.
在这个实现中，值 1.0 用作初始猜测。下图显示了程序收敛到开区间(- 2,2)中值的精确解所需的迭代次数。(实现不会收敛到这个范围之外的输入)。注意收敛速度取决于解决方案与初始猜测的接近程度。

13
13



What you need to do:
你需要做的:

1. Build and run cuberoot. Report the ISPC implementation speedup for a single CPU core (no tasks) and when using all cores (with tasks). What is the speedup due to SIMD parallelization? What is the speedup due to multi-core parallelization?
建立并运行 cuberoot。报告单个 CPU 核心(无任务)和使用所有核心(有任务)的 ISPC 实现加速情况。SIMD 并行化的加速度是多少？多核并行的加速度是多少？

2. Modify the function initGood() in the file data.cpp to generate data that will yield a very high relative speedup of the ISPC implementations. (You may generate any valid data.) Describe why these input data will maximize speedup over the sequential version and report the resulting speedup achieved (for both the with- and without-tasks ISPC implementations). You can test this version with the command-line argument “--data g.” Does your modification improve SIMD speedup? Does it improve multi-core speedup? Please explain why.
修改 data.cpp 文件中的函数 initGood ()来生成数据，这将产生一个非常高的 ISPC 实现的相对加速。(你可以生成任何有效的数据描述为什么这些输入数据将最大化连续版本的加速，并报告实现的结果加速(对于有任务和没有任务的 ISPC 实现)。你可以用命令行参数“—— data g”来测试这个版本。你的修改是否提高了 SIMD 的加速？它是否提高了多核加速？请解释原因。

3. Modify the function initBad() in the file data.cpp to generate data that will yield a very low (less than 1.0) relative speedup of the ISPC implementations. (You may generate any valid data.) De-scribe why these input data will cause the SIMD code to have very poor speedup over the sequential version and report the resulting speedup achieved (for both the with- and without-tasks ISPC imple-mentations). You can test this version with the command-line argument “--data b.” Does your modification improve multi-core speedup? Please explain why.

修改 data.cpp 文件中的函数 initBad ()， 以生成 ISPC 实现的相对加速度很低(小于 1.0)的数据。(你可以生成任何有效的数据描述为什么这些输入数据会导致 SIMD 代码相对于顺序版本的加速度非常差，并报告由此实现的加速度(对于有任务和没有任务的 ISPC 实现)。你可以用命令行参数“—— data b”来测试这个版本。你的修改是否提高了多核加速？请解释原因。

Notes and comments: When running your “very-good-case input”, take a look at what the benefit of multi-core execution is. You might be surprised at how high it is.
注释和评论: 当运行“非常好的案例输入”时，看看多核执行的好处是什么。你可能会对它的高度感到惊讶。

What you need to handin:
你需要递交的材料:

1. Provide explanations and answers in your report.
在你的报告中提供解释和答案。

2. The archive file you submit will contain your version of the file data.cpp.
您提交的归档文件将包含您版本的文件 data.cpp。

14

Hand-in Instructions
交上来的指示

As mentioned, you should have your report in a file report.pdf in the home directory for the assignment. (Make sure the report includes your name and Andrew Id.) In this directory, execute the command
如前所述，你应该把你的报告放在 report.pdf 文件夹中，在主目录下进行作业。(确保报告包括你的名字和安德鲁 Id 在这个目录中，执行命令

```
make handin.tar
Make handin.tar
```

This will create an archive file with your report, and some of the files you modified for the different problems.
这将为您的报告创建一个存档文件，以及您针对不同问题修改的一些文件。

Completing the handin then depends on your status in the class:
然后完成 handin 取决于你在课堂上的状态:

1. If you are a registered student, you should have a course Autolab account. Go to

   Autolab at: https://autolab.andrew.cmu.edu/courses/15418-s22

   如果你是一名注册学生，你应该有一个课程 Autolab 帐户

   and submit the file handin.tar.
   然后提交文件 handin.tar。

   Important: You should also submit the PDF version of your report (one file only: report.pdf) to
   重要提示: 你也应该将你的报告的 PDF 版本(只有一个文件: report.PDF)提交给
   Gradescope. You can do so at:
   你可在以下网址查阅:

   https://www.gradescope.com/courses/355470
   Https://www.gradescope. com/courses/355470

2. If you are on the waitlist, you will not have an Autolab account. Instead, you should run the provided program submit.py as follows:
   如果您在候补名单上，您将没有一个自动化实验室帐户。相反，你应该运行提供的程序 submit.py，如下所示:

   ```
   ./submit.py -u          AndrewID
   ./submit.py-u andrew
   ```

   where AndrewID is your Andrew ID.
   安德鲁就是你的安德鲁身份证。

In either case, you can submit multiple times, but only your final submission will be graded, and the time stamp of that submission will be used in determining any late penalties.

在任何一种情况下，你可以多次提交，但只有你的最终提交将被评分，并且该提交的时间戳将用于确定任何逾期罚款。