Question 2:

The code on Mandelbrot provided (included in a separate file) computes the Mandelbrot fractal image, achieving great speedups compared to a sequential execution, by utilizing both the CPU's cores and the SIMD execution units within each core.

ISPC language constructs 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 CPU's collection of parallel execution resources as efficiently as possible.

Program 2 –

Part 1. A Few ISPC Basics (10 points)

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 group of concurrent ISPC program instances (referred to as gang). The gang of instances runs to completion, then control returns back to the calling C code.

Please familiarize yourself with ISPC language constructs by reading through the ISPC walkthrough available at http://ispc.github.com/example.html.

The example program in the walkthrough is close to the implementation of mandelbrot_ispc() in mandelbrot.ispc.

Part 1 deliverable: Carefully inspect the code for mandelbrot ispc (without tasks) quoted above and write a 1-2-page summary on exactly how it operates, with details for each function.

Solution:

Summary

Let us see the functions which are being implemented in the ISPC implementation of Mandelbrot fractal image.

Functions

a. mandelbrotSerial

This function executes the algorithm in a serial manner like our normal sequential execution of programs.

b. mandelbrotThread

This function executes the algorithm in a parallel manner using threads

c. verifyResult

This is just to verify the results of the algorithm i.e. the output buffers produced using various strategies of being the serial only, using ISPC without tasks and with Tasks.

d. scaleAndShift

This is for scaling and shifting the image in question.

e. Main

This is a driver program that drives the various functions like mandelbrot serial , mandelbrot ispc , mandelbrot ispc with tasks and then compares the results with the normal serial execution and performs the write functions.

f. mandelbrot_ispc

This function executes the algorithm using the SIMD capabilities through ISPC For each construct over all 2D array of image.

g. mandelbrot_ispc_withtasks

This function executes the algorithm using the SIMD capabilities and multi core architecture of the CPU and hence dividing the 2D array image into groups of rows where the number of groups is the number of tasks to be launched in separate cores of the CPU hence increasing the level of parallelism in the execution of the algorithm .

Now let us look at the important functions which form an integral part of the algorithm and important to see the advantage of parallel execution using the ISPC compiler.

h. mandel

This is the major function/set of instructions that work on each pixel and then return a number based on the set logic in the function.

Main thing behind the Mandel Fractal image algorithm is that we have to go over each pixel having x,y coordinates and hence can be looked like a complex number x+jy and perform a certain set of instructions.

What do these set of instructions do ?

We iterate till the maximum number of iterations given or until the distance of the given coordinate from the origin ($x^2 + y^2 > 4$) is met and the number of iterations till we have met either of the conditions is being returned.

In each iteration, we take the initial vector (X, Y) of the pixel as a constant and then from this initial vector (X,Y) we compute its distance from origin i.e. the x^2+y^2 and check if it is less than 4 and if yes then we compute the square of the complex number x+jy and add constant to this calculated square and this $z^2 +C$ will act as a new vector to work on for next iteration. We continuously perform these tasks for various iterations until we met the exit conditions or reach the maximum number of iterations.

Exact number of iteration where end condition is met is returned as a number which is written in the output buffer corresponding to the input pixel.

The interesting thing about this algorithm is that we have to execute mandel function as described above over all the pixels and here using the ISPC compiler we have performed through the different strategies:

1. Serial

2. Through for Each construct using SIMD capabilities

3. Through for Each construct and launch feature to launch various tasks to be executed on various cores.

There are three functions in the code for each of the above. Though the underlying logic is the same , let us look at the functions and their underlying differences.

1. Serial

2. ISPC

3. ISPC with launch tasks

Common Thing being performed is:

1. They are calculating the step value , which we can see as the interpixel distance normalized through the height or width in the corresponding direction.

2. Then they are looping through the image to every pixel , then calculating the vector (X, Y) (x0 + i * dx , y0 + j * dy) for each pixel or coordinate and then calculating the output index (note that buffer is linear and hence need to calculate a output index corresponding to pixel x,y coordinates) for writing the output return from mandel function.

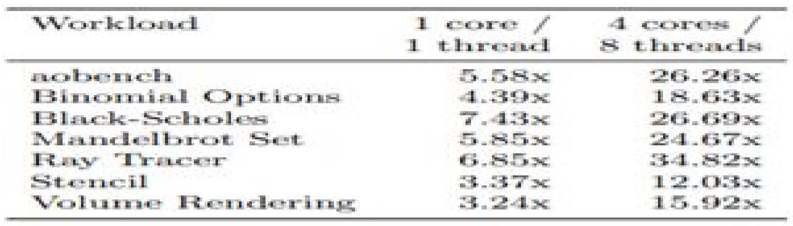
Now the difference in the 1 and 2 implementation is that 1 has two loop as in sequential execution to reach each pixel and perform **mandel function** while in the 2 implementation we are utilizing the algorithm characteristic of having implicit parallelism and hence utilizing the SIMD capability we are converting into the gang of instances and hence executing in parallel on various pixels (depends on the SIMD capability) in parallel all using the ISPC powerful feature of FOR EACH LOOP which implicitly does all these instance creation stuff and execute our program as SPMD using SIMD feature.

1) What is the maximum speedup you expect in theory (and ideally) given the specifications of your allocated CPUs?

Theoretical speed up = SIMD width of the core (4 or 8 generally) but practically it is seen to be about 5.5 times.

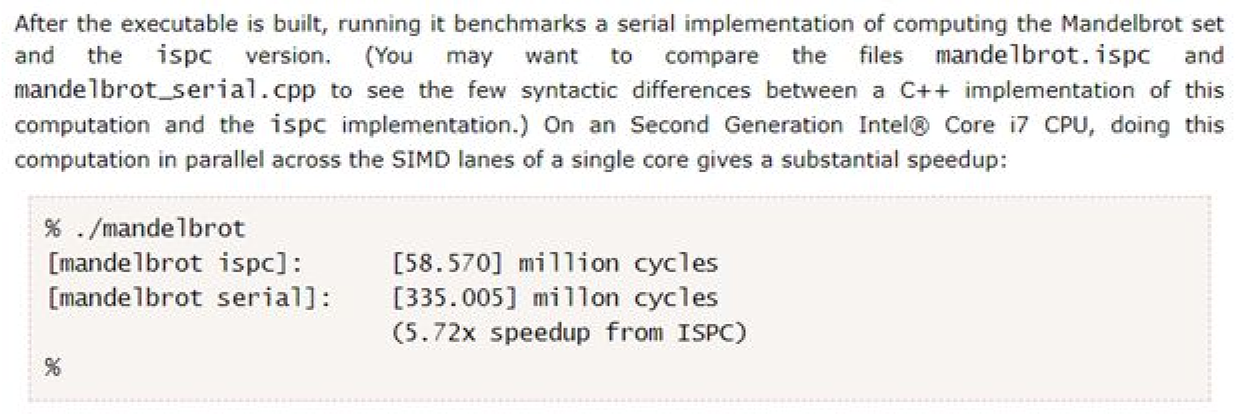
This is being said based on 1 core 8 width SIMD.

Note that the ISPC gang abstraction used **for each** is implemented by SIMD instructions on one core.



Workload	1 core / 1 thread	4 cores / 8 threads
aobench	5.58x	26.26x
Binomial Options	4.39x	18.63x
Black-Scholes	7.43x	26.69x
Mandelbrot Set	5.85x	24.67x
Ray Tracer	6.85x	34.82x
Stencil	3.37x	12.03x
Volume Rendering	3.24x	15.92x

Table 2: Speedup of various workloads on a single core and on four cores of a system with 8-wide SIMD units, compared to a serial C++ implementation. The one core speedup shows the benefit from using the SIMD lanes of a single core efficiently, while the four core speedup shows the benefit from filling the entire processor with useful computation.



After the executable is built, running it benchmarks a serial implementation of computing the Mandelbrot set and the `ispc` version. (You may want to compare the files `mandelbrot.ispc` and `mandelbrot_serial.cpp` to see the few syntactic differences between a C++ implementation of this computation and the `ispc` implementation.) On an Second Generation Intel® Core i7 CPU, doing this computation in parallel across the SIMD lanes of a single core gives a substantial speedup:

```
% ./mandelbrot
[mandelbrot ispc]:      [58.570] million cycles
[mandelbrot serial]:    [335.005] millon cycles
                        (5.72x speedup from ISPC)
%
```

Here are some results from the research paper and the intel walkthrough example.So single core SIMD leading to above mentioned results.

2) Why might the number you observe be less than this ideal (take our word for this or ... do the extra bonus questions and make the proper modifications to compile it and run it yourself)?

Choosing A Target Vector Width or Targeted instruction set

By default, ispc compiles to the natural vector width of the target instruction set. For example, for SSE2 and SSE4, it compiles four-wide, and for AVX, it complies 8-wide. For some programs, higher performance may be seen if the program is compiled to a doubled vector width--8-wide for SSE and 16-wide for AVX.

For workloads that do not require many registers, this method can lead to significantly more efficient execution thanks to greater instruction level parallelism and amortization of various overhead over more program instances. For other workloads, it may lead to a slowdown due to higher register pressure; trying both approaches for key kernels may be worthwhile. This option is only available for each of the SSE2, SSE4 and AVX targets. **It is selected with the --target=sse2-x2, --target=sse4-x2 and --target=avx-x2 options, respectively.**

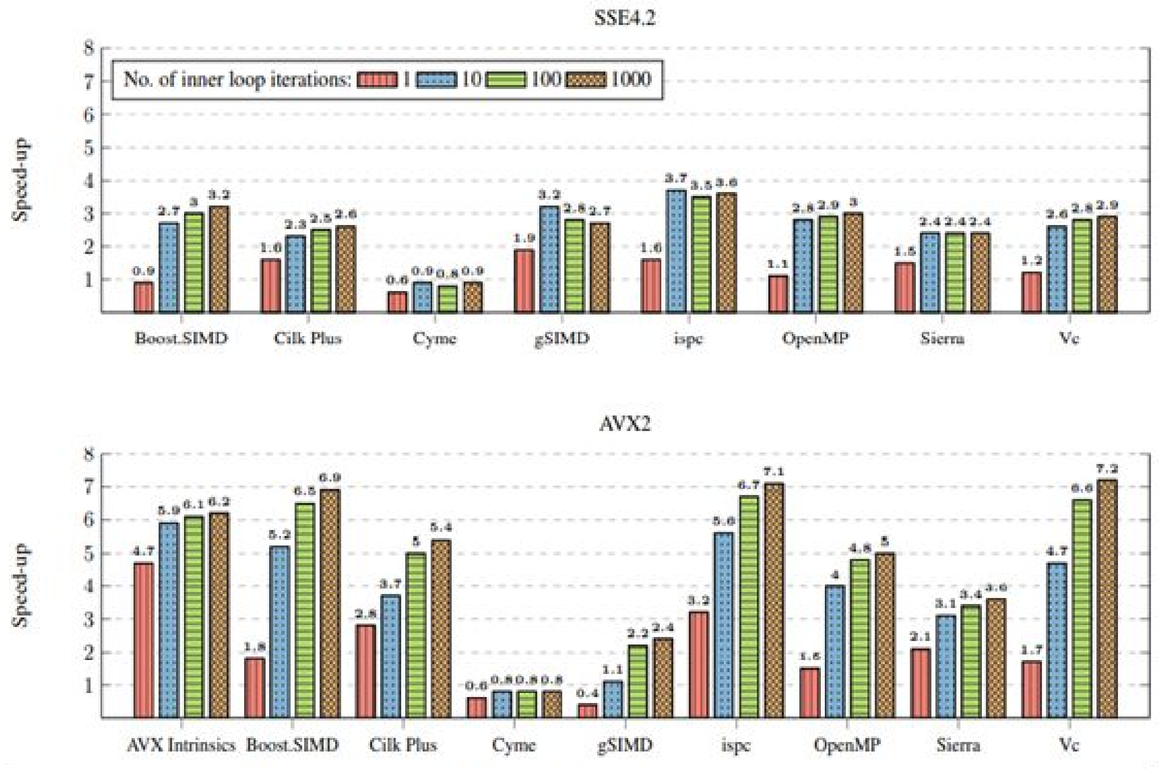


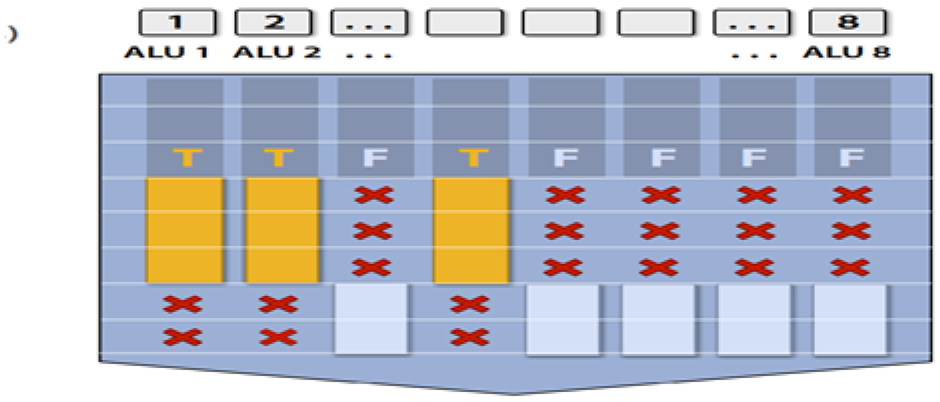
Figure 3. Acceleration of a Mandelbrot kernel with SSE4.2 and AVX2 SIMD units, shown for varying iterations of the inner `while`-loop

The above graph we found from the internet search , where the Mandelbrot kernel producing the SSE and AVX2 was compared. As we can see there is a performance difference when the compiler targets different instructions AVX or SSE. So this point is very important while executing using the ISPC.

3) Consider the characteristics of the computation you are performing. Describe the parts of the image that present challenges for SIMD execution. Why is this so?

The main characteristics of thee computation involved is that on each pixel we are performing floating point calculations involving various iterations of Z^2 + C at every pixel where c is x, y of the pixel which also increases when pixels are on higher dimensional side of the image(can be visualized as the bottom right corner) so the computations start becoming heavy in those parts.(so it can be challenge as we discussed in the issue in the class about why interleaved assignments are better)

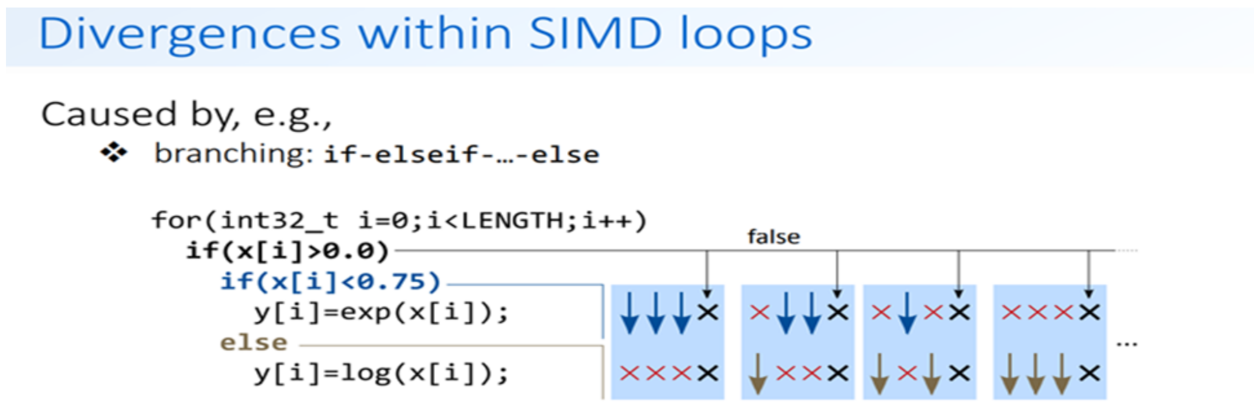
Second is in the parts of the image where suppose the x2 + y2 > 4 as mentioned here is reached and other elements in the vector are not able to satisfy so the performance enhancement is not that balanced. This issue was also discussed in the class about the control flow situations where one satisfies the if condition and other not. So this is also a road block for SIMD execution where one pixel or Data point is dragging the computation of other elements.



Not all ALUs do useful work!
Worst case: 1/8 peak performance

4) What adverse phenomenon do these characteristics add to the execution (what is the name of this phenomenon)?

As explained in the slides and class the best use cases of SIMD are when there are coherent execution and not when there is a divergent execution.And in our case, it seems to be divergent execution.

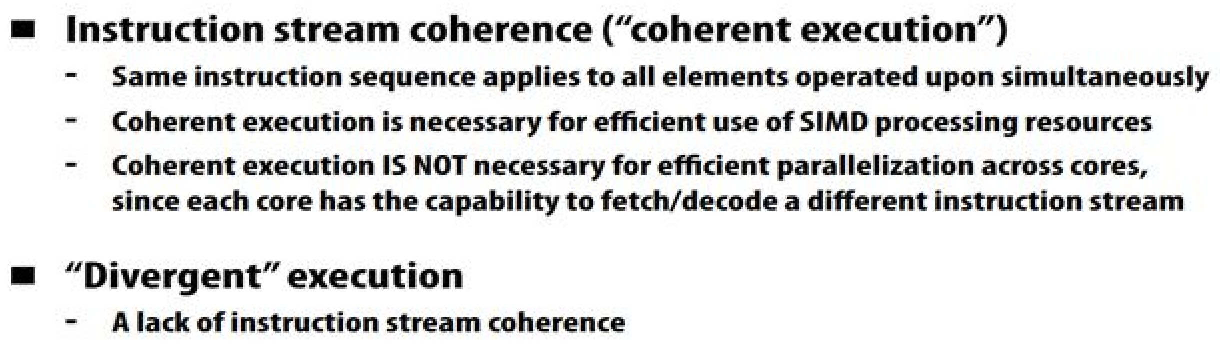


Divergences within SIMD loops

Caused by, e.g.,

- branching: `if-elseif-...-else`

```
for(int32_t i=0;i<LENGTH;i++)
  if(x[i]>0.0)
    if(x[i]<0.75)
      y[i]=exp(x[i]);
    else
      y[i]=log(x[i]);
```


- **Instruction stream coherence ("coherent execution")**
  - **Same instruction sequence applies to all elements operated upon simultaneously**
  - **Coherent execution is necessary for efficient use of SIMD processing resources**
  - **Coherent execution IS NOT necessary for efficient parallelization across cores, since each core has the capability to fetch/decode a different instruction stream**
- **"Divergent" execution**
  - **A lack of instruction stream coherence**

Note: for such a piece of code, the ISPC compiler maps gangs of program instances to SIMD instructions executed on a single core.

Extra Bonus! (TBD points): Compile and run the program mandelbrot ispc after you make the proper changes to set it up and adjust it to your environment. CAUTION: This may take some time and it is for those of you that are curious, hands-on, and eager enough to attempt!

Program 2 - Part 2. ISPC Tasks (8 points)

ISPCs SPMD execution model and mechanisms like foreach facilitate the creation of programs that utilize SIMD processing. The language also provides an additional mechanism utilizing multiple cores in an ISPC computation. This mechanism is launching ISPC tasks. The launch[2] command in the function mandelbrot_ispc_with tasks launches two tasks. Each task defines a computation 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. Similar 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).

Part 2 deliverable: Study how mandelbrot_ispc can be run with the parameter --tasks.

Answer:

In order to make it easy to fill multiple CPU cores with computation, ispc provides an asynchronous task launch mechanism, closely modeled on the “spawn” facility provided by others. ispc functions called in this manner are semantically asynchronous function calls that may run concurrently in different hardware threads than the function that launched them. This capability makes multi-core parallelization of ispc programs straightforward when independent computation is available; generally, just a few lines of additional code are needed to use this construct.

In the implementation of mandelbrot_ispc with Tasks feature , where two tasks are launched so in our present case as mentioned above that our tasks can run concurrently in different hardware threads so our program will speed up to existing mandelbrot_ispc function by 2.

As given by the function `mandelbrot_ispc_task`, each task computes a region of the final image. Similar 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). This type of problem decomposition is referred to as spatial decomposition since different spatial regions of the image are computed in parallel manner.

Study the ispc walkthrough available (or additional resources you may find) and come up with an approximation for the speedup using tasks over the version of mandelbrot_ispc that does not partition that computation into tasks.

Question 1: Does the speedup depend on the parameter tasks and in what way? The performance of mandelbrot_ispc --tasks can be improved by changing the

number of tasks the code creates. By only changing code in the function mandelbrot_ispc_withtasks(), you should be able to achieve performance that exceeds the sequential version of the code by a great deal!

Answer: As mentioned , ispc functions made tasks and called in this manner (using launch with the number as here is 2), are semantically asynchronous function calls that may run concurrently in different hardware threads than the function that launched them. This capability makes multi-core parallelization of ispc programs straightforward when independent computation is available.

So as we have independent computation and hence making the number of tasks as 2 , we are processing the two sections of the image parallely using tasks launch feature and hence theoretical speed up of 2. So yes, we can say that speed up happens with an increase in the number of tasks. But we can not just increase the tasks and get the speed up. It depends on the cores and the architecture of the processing unit. And yes, by increasing the tasks or we can say the kind of software threads , they can hide various I/O , Memory latencies but they come at the cost of context switching , scheduling times and communication etc. overheads. So, it is necessary to evaluate the working of our algorithm/computation, architecture available CPU , cache etc. and then decide on the granularity of tasks.

Question 2: How do you determine in theory how many tasks to create? Why do you think that the number you chose works best? Provide justification by theory or other counter examples.

We would start by mentioning that Granularity affects the performance of parallel computers. Using fine grains or small tasks results in more parallelism and hence increases the speedup. However, synchronization overhead, scheduling strategies etc. can negatively impact the performance of fine-grained tasks. Increasing parallelism alone cannot give the best performance.

So considering the above let us understand our problem in hand that considering we have 4 core CPUs without hyperthreading , so logical core and hardware cores are the same then in that case it would be appropriate to divide our image data into the 4 tasks that can be executed in parallel. Hence if we chose 2 we get almost that amount of speed up to non-task ispc considering we have more than one core available.

But dividing into tasks or launching tasks is like launching the parallel software threads for a given architecture.

One thing we have for our computation is that the floating-point computation for each pixel or data point is completely independent of the other so there are no synchronization or communication issues. So, in that case as mentioned on first sight we can make our tasks as granular as we can but there are some issues with it like scheduling overhead , context switching (synchronization/communication not at present and cores are also limited). So , considering a situation where the data is very large to work on and then in that case for every core , the required data will not come on the cache to work on and in that case latency will happen in getting the data again. So, in such scenarios if we have more software threads or tasks in ispc case, then we can have latency hiding.

In our case , where the dataset is not that big , it is felt that the ideal will be to create that many tasks that can run in parallel on all Cores or Hardware Threads simultaneously.

So theoretically it is felt that –

Graph of speed up will increase with increase in tasks will reach the peak and then start decreasing.

And peak is decided by the architecture(number of cores etc.) , memory intensity of the tasks so as to decide what number of tasks will be appropriate to enhance overall performance.

Extra Bonus! (TBD points): Compile and run the program mandelbrot ispc after you make the proper changes to adjust it to your environment with the parameter --tasks. What speedup do you observe? What is the speedup over the version of mandelbrot_ispc that does not partition that computation into tasks? By only *changing code in function mandelbrot_ispc_withtasks(), you should be able to achieve performance that exceeds the sequential version of the code by a great deal! By how exactly for your own ORBIT cores set-up? How do you determine how many tasks to create? CAUTION: This may take some time and it is for those of you that are curious, hands-on, and eager enough to attempt despite the time limitations!*