ECE 566 Parallel and Distributed Computing, Rutgers University, Fall 2020

Instructor: Maria Striki

Programming Project 1: (22 + 18 + 32 = 72 points):

Issued Date: Fri Oct 9th 2020, Due Date: Thu Oct 22nd, 9.00 pm.

To be conducted in groups of 1-3. When you run your programs, you may compare them against other nodes in your group, if the specifications of other machines (nodes) are different. Again, please produce an extensive report that displays the results of your experimentation, describes your code, and answers all questions addressed.

*** **IMPORTANT:** Groups that consist of only one member please skip Problem 1. Or I can allow groups with low number like 1 or 2 members to work together for this programming project.

Problem 1: Experimenting with your machine's memory hierarchy and obtaining latencies and throughput (22 points)

Design an experiment (i.e., design, sketch, document your experiment method, and write simple programs and take the proper measurements) in order to determine the memory bandwidth of your computer and estimate caches at various levels of hierarchy). Your computer does have L1 and may have L2 or L3. So, you may safely measure your L1 size and then conjecture if you have an L2.

To test cache size, you want to apply data re-use, note down the effect of the cache (in seconds, using a time function) and then you want to increase the cache size until the reuse decreases sharply.

To test bandwidth, you want to take measurements when you do not have data re-use.

A good approximation to the bandwidth and to L1 cache (and potentially more levels) can be obtained from one or more (nested) loops that execute certain instructions. You must identify in your design what sort of instructions these may be and what sort of data they handle to approximate the two distinct parameters described above.

You will need but you are also expected to plot your results. Your plots will be a function of your data size, and determine increase or sudden drops in performance, and also determine where these drops occur.

Hint: Please note that a very simple mathematical operation that takes potentially one clock cycle, such as the addition, may be ignored if it is coupled with a memory bound operation.

Problem 2: Parallel Fractal Generation -- Investigation (18 points)

Question: 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. Then, by connecting the problem to your group's allocated ORBIT Cores specifications **answer the following questions**:

1) What is the maximum speedup you expect in theory (and ideally) given the specifications of your allocated CPUs?
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)?
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?
4) What adverse phenomenon do these characteristics add to the execution (what is the name of this phenomenon)?

Note: for such 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_withtasks` 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. 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!

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

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!

Problem 3: Parallel Square Root Computation (32 points)

Question: Write an ISPC program that computes the square root of 20 million random numbers (real, no necessarily integers) between 0 and 9 (*Hint*: Use Newton's method to solve the equation: $1/x^2 - S = 0$). Please use multiple iterations for sqrt to converge to an accurate solution (accurate solution is one that gives < 10^-4 difference compared to the true value).

ISPCs SPMD execution model (and libraries), like "foreach", implement SIMD processing. ISPC also provides a mechanism that "launches ISPC tasks", utilizing this way multiple cores in an ISPC computation. Check the launch[*num*] command in function mandelbrot_ispc_withtasks and in the Intel ISPC Spec. This command launches *num* tasks. Each task defines a computation that will be executed by a "thread" of ISPC gangs/instances. As given by the function mandelbrot_ispc_task, each task computes a region of the final image in any order (in parallel on different CPU cores).

For ease of your understanding how to handle the code, I am also attaching a README file that gives you an idea how to use all the files you will generate into the ORBIT system and/or any LINUX OS without generating a Makefile. Of course, you are welcome to build a Makefile.

You will additionally need to use a number of existing system files, such as: tasksys.cpp

I am also including two related .ispc and .h files to help you direct the rest of your coding: `square_root_serial.h` and `square_root.ispc.`

You are welcome to modify those files in order to match with the rest of your coding. You would need to write a `square_root_tasks.ispc`, a `square_root_avx.c` , of course a `square_root_main.cpp`. You need to randomly generate your input and modify the number of elements and the number of tasks at run time from the user prompt in order to conduct timing tests w.r.t : method (serial vs. only one core vs. multiple cores) and number of elements and number of tasks used.

Deliverables:

1) What is the ISPC implementation speedup for single CPU core (no tasks launched) and when using multiple cores (with tasks)?
2) What is the speedup due to SIMD parallelization?
3) What is the speedup due to multi-core parallelization?
4) Extend your code with the potential to utilize 2, 3, 4, 5, 6, 7, 8 threads, partitioning the computation accordingly. In your write-up, produce a graph of speedup compared to a sequential implementation (**which you need to implement as well**) as a function of the number of cores and threads used.
5) Is your speedup linear in the number of cores used? In the number of threads used? Please justify why yes or why not. What is the relation of your speedup w.r.t. to the specifications of your machine?
6) Then **write another version of the sqrt function using AVX intrinsics**. You may consult the Intel Intrinsics Guide: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

If you wish to tell the ISPC compiler to generate AVX, rather than SSE instructions, please see the Makefile comment about ISPC's --target=avx2-i32x8 compiler flag. For slightly better performance use --target=avx2-i32x16, which will use a gang size of 16 instances and use two AVX instructions to implement an operation for the entire gang. (more details here: https://ispc.github.io/ispc.html#selecting-the-compilation-target).

Extra clarification on the last questions of Part 1 of your ISPC project:

Implement the AVX manually by AVX intrinsics, compare it with the automatic implementation which is within the ISPC (SSE4), to see how much difference in performance can be identified between them. So there is no need to use ISPC to compile the avx intrinsics, GCC is enough.

Normally, you should do for avx sqrt exactly what you did for ispc, so you should be launching tasks. If we are to compare performances, we need to be very rigorous about implementing the avx and the ispc-sse4 in a similar way. Otherwise, the performance comparison makes no sense.

You are encouraged to run your code on the orbit testbed in WINLAB. Use your ORBIT account per group. Time slots are reserved everyday normally between 20.00-12.00am but also randomly earlier in the day. The ISPC compiler is already installed in "mariasfirstimage" but you may download ISPC at http://ispc.github.com/.