



TÉCNICO
LISBOA

Assignment 1

(part 2)

High Performance Computing Systems

Sistemas de Computação de Elevado Desempenho (SCED)

2020/2021

Overview

This assignment is intended to help you develop an understanding of the two primary forms of parallel execution present in a modern multi-core CPU:

1. SIMD execution within a single processing core
2. Parallel execution using multiple cores.

You will also gain experience measuring and reasoning about the performance of parallel programs (a challenging, but important, skill you will use throughout this class). This assignment involves only a small amount of programming, but a lot of analysis!

Deadlines and Submission

You will submit both parts of Lab1 via Fenix (until 18th of October at 23h59, i.e., Sunday).

The submission should be made in a single zip file with the following content:

- Your report (writeup) in a file called `report.pdf` (it must be pdf)
- Your implementation (codes) for all programs (please do make clean before submission)

Your report and codes should correspond to both parts of Lab1. When handed in, all codes must be compilable and runnable out of the box on the adriana machine!

Your report should have at most 10 pages and should be well structured. For each problem (program) you should: elaborate your solution, explain your rationale (how did you arrive there, why it makes sense, describe all (if any) specific measurements/experiments that were conducted by you to prove your hypothesis etc). You should also discuss and analyze the obtained results!

Environment Setup

For this assignment, you will need to run (the final version of) your code on adriana machine (hostname: `adriana.inesc-id.pt`). This machine contains four-core 3.5 GHz Intel Core i7 processor (although dynamic frequency scaling can take it to 3.9 GHz when the chip decides it is useful and possible to do so). Each core in the processor can execute AVX2 vector instructions which describe simultaneous execution of the same operation on multiple single-precision data values. For the curious, a complete specification for this CPU can be found at <https://ark.intel.com/.../intel-core-i7-3770k-processor-8m-cache-up-to-3-90-ghz.html>.

The access to adriana machine is performed through ssh, by using the command:

```
ssh HPCS_g<group_number>@adriana.inesc-id.pt
```

where you should substitute `<group_number>` with your group number (e.g., the username for the group number 3 is `HPCS_g3`). The passwords for each group have the flowing format `g<group_number>hpcs` (again, the password for the group number 3 is `g3hpcs`).

IMPORTANT: Since the machine is a shared resource, please verify if nobody else is performing experiments before running your program, in order not to affect the results of other groups (or even other researchers from INESC-ID). To check the users logged in the machine, you can use the **who** command. It is highly recommended to schedule the utilization of adriana among yourselves to avoid the simultaneous use of the machine (you should also think why it is not a good idea that all of you run the programs at the same time – given the characteristics of adriana’s CPU – and how it can impact the execution of your program).

Note: For grading purposes, we expect you to report on the performance of code run on the adriana machine. However, for development, you may also want to run the programs in this assignment on your own machine. Feel free to include your findings from running code on other machines in your report as well, just be very clear what machine you were running on.

Note (Lab1,part 2): If you are using your own machine, you will first need to install the Intel SPMD Program Compiler (ISPC) available here: <http://ispc.github.com/>

To get started:

1. ISPC is needed to compile many of the programs used in this assignment. ISPC can be easily installed on your Linux machine through the following steps (you don’t need to this on adriana, since ISPC is already installed there):
 - Download the linux binary into a local directory of your choice. You can get ISPC compiler binaries for Linux from the ISPC [downloads page](#). You can use wget to directly download the binary from the downloads page:
`wget https://github.com/ispc/ispc/releases/download/v1.14.1/ispc-v1.14.1-linux.tar.gz`
 - Untar the downloaded file: `tar -xvf ispc-v1.14.1-linux.tar.gz`
(You may need to change the ISPC version number in the filename accordingly.)
 - Add the ISPC directory to your system path. For example, if untarring the downloaded file produces the directory `~/Downloads/ispc-v1.14.1-linux`, in bash you'd update your path variable with:
`export PATH=$PATH:${HOME}/Downloads/ispc-v1.14.1-linux/bin`
The above line can be added to your `.bashrc` file for permanence. If you are using `csh`, you'll update your PATH using `setenv`. A quick Google search will teach you how.
2. The assignment starter code is available on the course webpage (section: Labs).

Problem 2: Vectorizing Code using our “fake” SIMD Intrinsics

Take a look at the function `clampedExpSerial` provided below. The `clampedExp()` function raises `values[i]` to the power given by `exponents[i]` for all elements of the input array and clamps the resulting values at 9.999999. In Problem 2, your job is to vectorize this piece of code, so it can run on a machine with SIMD vector instructions.

```
// For each element, compute values[i]^exponents[i] and clamp value to 9.999.
// Store result in output.
void clampedExpSerial(float* values, int* exponents, float* output, int N) {
    for (int i=0; i<N; i++) {
        float x = values[i];
        int y = exponents[i];
        if (y == 0) {
            output[i] = 1.f;
        } else {
            float result = x;
            int count = y - 1;
            while (count > 0) {
                result *= x;
                count--;
            }
            if (result > 9.999999f) {
                result = 9.999999f;
            }
            output[i] = result;
        }
    }
}
```

However, rather than craft an implementation using SSE or AVX2 vector intrinsics that map to real SIMD vector instructions on modern CPUs, to make things a little easier, we're asking you to implement your version for our “fake” processor” using our “fake vector intrinsics” from theoretical classes and run in on the paper ☺

Here are few hints to help you in your implementation:

- Consider that the total number of loop iterations is a multiple of SIMD vector width.
- Consider the following input values:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
values	3.0	2.0	2.5	1.25	5.5	0.5	10.1	3.15	1.75	6.55	1.63	1.5	4.33	0.15	1.95	2.83
exps.	0	2	3	10	0	4	0	3	5	1	4	9	0	5	0	3

- *Hint:* For this exercise `_vpopcnt` (popcount) intrinsic is introduced in our "fake vector intrinsics". Popcount instructions count the number of set bits in a value. Using the `_vpopcnt` intrinsic on `__vbool` data may be helpful in this problem.

You should consider the performance of your implementation to be the value "Total Vector Instructions". (You can assume every "fake" vector instruction takes one cycle on the fake SIMD CPU.) You can also calculate the "Vector Utilization", i.e., the percentage of vector lanes that are enabled during the execution of your code.

What you need to do:

1. Implement a vectorized version `clampedExpVector` of `clampedExpSerial`. Calculate the "Total Vector Instructions" and "Vector Utilization" metrics for the given input values. Try to visually represent the execution of your code (for the first 2 loops) by considering the vector width of 8 (focus on vector lanes utilization -- slides from theoretical classes might be helpful to get some ideas).
2. Consider that we change the vector width from 2, 4, 8, to 16. Does the vector utilization increase, decrease or stay the same as the vector width changes? Why?

Program 3: Parallel Fractal Generation Using ISPC

Like Program 1 from part 1, Program 3 computes a mandelbrot fractal image, but it achieves even greater speedups by utilizing both the CPU's four cores and the SIMD execution units within each core.

In Program 1 (Lab1, part 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. (Since threads were one-to-one with processing cores in Program 1, you effectively assigned work explicitly to cores.) Instead of specifying a specific mapping of computations to concurrently executing threads, Program 3 uses ISPC language constructs to describe *independent computations*. These computations may be executed in parallel without violating program correctness (and indeed they will!). 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.

You will make a simple fix to Program 3 which is written in a combination of C++ and ISPC (the error causes a performance problem, not a correctness one). With the correct fix, you should observe performance that is over twenty times greater than that of the original sequential Mandelbrot implementation from `mandelbrotSerial()`.

Program 3, Part 1. ISPC Basics

Before proceeding, you are encouraged to 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 almost exactly the same as Program 3'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.

What you need to do:

1. Compile and run the program `mandelbrot_ispc`. **The ISPC compiler is currently 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? Describe the parts of the image that present challenges for SIMD execution? Comparing the performance of rendering the different views of the Mandelbrot set may help confirm your hypothesis.)

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 Program 1, where speedup was achieved by running threads on multiple cores.

Program 3, Part 2: ISPC Tasks

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*.

See the `launch[2]` command in the function `mandelbrot_ispc_withtasks`. This command launches two tasks. 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. 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).

What you need to do:

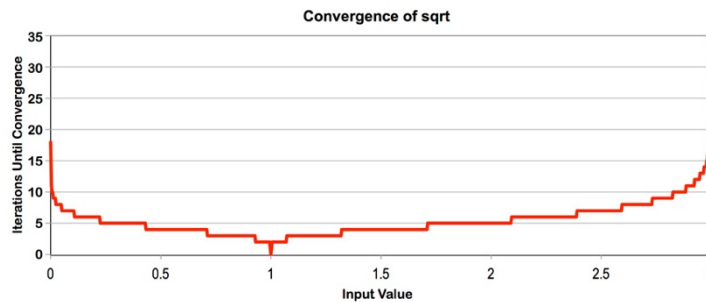
1. Run `mandelbrot_ispc` with the parameter `--tasks`. What speedup do you observe on view 1? What is the speedup over the version of `mandelbrot_ispc` that does not partition that computation into tasks?
2. There is a simple way to improve the performance of `mandelbrot_ispc --tasks` by changing the 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 over 25 times! How did you determine how many tasks to create? Why does the number you chose work best?

Program 4: Iterative sqrt

Program 4 is an ISPC program that computes the square root of 20 million random numbers between 0 and 3. It uses a fast, iterative implementation of square root that uses Newton's method to solve the equation $1/x^2 - s = 0$. This gives a value $x \approx \sqrt{1/s}$. Multiplying x by s gives an approximation to \sqrt{s} .

The value 1.0 is used as the initial guess in this implementation. The graph below shows the number of iterations required for `sqrt` to converge to an accurate solution for values in the (0-3) range. (The implementation does not converge for inputs outside this range). Notice that the speed of convergence depends on the accuracy of the initial guess.

Note: This problem is a review to double-check your understanding, as it covers similar concepts as programs 2 and 3.



What you need to do:

1. Build and run `sqrt`. Report the ISPC implementation speedup for 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?
2. Modify the contents of the array values to improve the relative speedup of the ISPC implementations. Describe a very-good-case input that **maximizes speedup over the sequential version of the code** and report the resulting speedup achieved (for both the with- and without-tasks ISPC implementations). Does your modification improve SIMD speedup? Does it improve multi-core speedup (i.e., the benefit of moving from ISPC without-tasks to ISPC with tasks)? Please explain why.
3. Construct a very-bad-case input for `sqrt` that **minimizes speedup for ISPC with no tasks**. Describe this input, describe why you chose it, and report the resulting relative performance of the ISPC implementations. What is the reason for the loss in efficiency? (**keep in mind we are using the `--target=avx` option for ISPC, which generates 8-wide SIMD instructions**).

Program 5: BLAS saxpy

Program 5 is an implementation of the `saxpy` routine in the BLAS (Basic Linear Algebra Subproblems) library that is widely used (and heavily optimized) on many systems. `saxpy` computes the simple operation $\text{result} = \text{scale} * X + Y$, where X , Y , and `result` are vectors of N elements (in Program 5, $N = 20$ million) and `scale` is a scalar. Note that `saxpy` performs two math operations (one multiply, one add) for every three elements used. `saxpy` is a *trivially parallelizable computation* and features predictable, regular data access and predictable execution cost.

What you need to do:

1. Compile and run `saxpy`. The program will report the performance of ISPC (without tasks) and ISPC (with tasks) implementations of `saxpy`. What speedup from using ISPC with tasks do you observe? Explain the performance of this program. Do you think it can be substantially improved? (For example, could you rewrite the code to achieve near linear speedup? Yes or No? Please justify your answer.)

Notes: Don't think too hard! We expect a simple answer, but the results from running this problem might trigger more questions in your head.