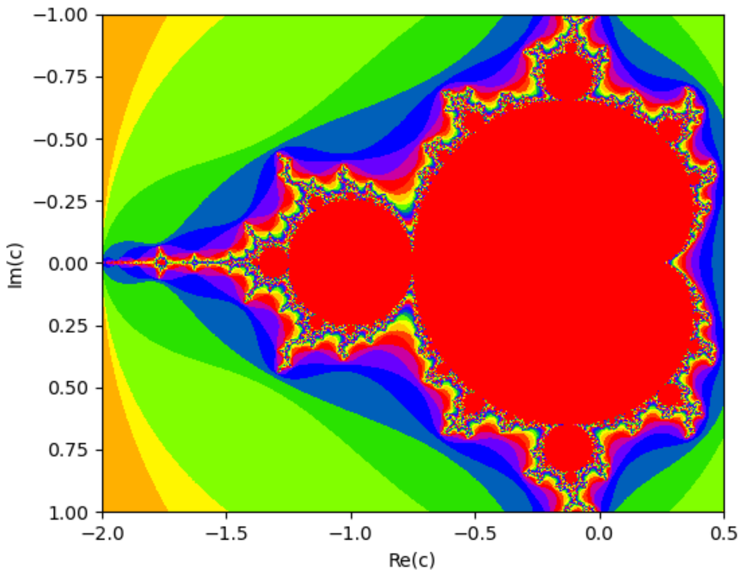
Exercise 5 : vectorization

Sébastien Ponce

October 6, 2021



1 Foreword

This exercise plays with an implementation of a mandelbrot set display. The display code itself is written in python in mandel.py and has been made generic by the usage of an underlying dynamic library, whose name is given on the command line.

E.g. running ‘python mandel.py‘ will make use of libmandel.so while ‘python mandel.py intr‘ will use libmandelintr.so. This allows to implement multiple backends for our computation, compiled in different libraries.

A scalar version of the computation code is provided in mandel.cpp and compiled to libmandel.so. Please have a look at the code. The main method (called from python, and thus with C linkage) is mandel. It essentially loops on the pixels of the image to generate and calls a ‘kernel‘ method for each of them.

The kernel method itself tries to find out if and how fast the recursion $z = z^2 + a$ diverges for a given z. It returns the iteration at which $|z|$ exceeds 2 (i.e. 4 for $|z|^2$) or -1 it it does not reach it within 100 iterations.

The goal of this exercise is to implement a vectorized backend, using the VCL library, and compile it to libmandelVCL.so. The idea is simple : compute a vector of pixels at each iteration rather than a single pixel.

2 Goals of the exercise

- take scalar piece of code and see how to vectorize it
- use godbolt to check for optimizations and vectorized code
- try VCL backend, others are given as examples

3 Setup

- open a terminal
- log in to the csc-cc-pm machine you were provided via ssh, with X support

```
ssh -X csc_<xxxxxx>@tcsc-2021-autumn-<nn>.cern.ch
```

- clone the exercise 5 code in /tmp/¡username¿, or reuse a previous clone. You're actually free to clone it somewhere else.

```
cd /tmp
mkdir <username>
cd <username>
git clone https://gitlab.cern.ch/sponce/tcsccourse.git
```

- setup the environment to use a gcc 11 as a compiler

```
source /cvmfs/sft.cern.ch/lcg/releases/gcc/11.1.0/x86_64-centos7/setup.sh
```

- go to exercise5 and compile the example code using cmake

```
cd tcsccourse/exercises/exercise5
make libmandel.so
```

- check that everything works fine and you get a nice mandelbrot set

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:.
python mandel.py
```

How much time did the computation take in scalar mode ?

4 Autovectorization attempt

Let's check whether the code can auto vectorize and understand the issues. Here the easiest is to use the godbolt web site. Go to `https://godbolt.org/z/PErqWhsdT`, inspect the generated code and convince yourself that nothing has been vectorized.

Check in particular the lines corresponding to the 'kernel' function (assembler lines 9 to 25).

What proves that we did not get vector instructions ?

In the top of the right panel, click on “Add New” and add an “Optimization output”. You can now see the code with colored patches on the left. Going over them will give you a lot of information on the optimizations the compiler could or could not do on your code.

Let’s concentrate on ‘kernel‘. Hovering the green part next to line 33 tells you it was inlined. Hovering the red next to line 9 tells us why that loop could no be vectorized. Out of the many messages, can you extract the key point making autovectorization impossible ?

Why is the compiler unable to vectorize the loop inside the ‘kernel‘ function ?

Note also the compiler saying “unrolled loop by a factor of 2 with a breakout at trip 0”. See what it means in the assembly code and how the loop was partially unrolled to gain a bit of performance.

5 Manual vectorization using VCL

The idea of this vectorization is simple : make an AVX specific version where we take the pixels 8 by 8 in the second for loop of the ‘mandel‘ function. In practice, we change :

```
float ay = miny;
for (unsigned int any = 0; any < ny; ay += dy, any++) {
  float ax = minx;
  for (unsigned int anx = 0; anx < nx; ax += dx, anx++) {
    buffer[any*nx+anx] = kernel(ax, ay);
  }
}
```

into :

```
Vec8f ay{miny}; // vector of 8 miny value
// loop one pixel at a time in y
for (unsigned int any = 0; any < ny; ay += dy, any++) {
  // vector of first 8 pixels in x
  Vec8f ax{minx, minx+dx, minx+2*dx, minx+3*dx,
    minx+4*dx, minx+5*dx, minx+6*dx, minx+7*dx};
  // loop 8 by 8 pixels in x
  for (unsigned int anx = 0; anx < nx; ax += 8*dx, anx += 8) {
    // compute and store back 8 pixels in one go
    kernel(ax, ay).store(buffer+any*nx+anx);
  }
}
```

Your task is to write the corresponding kernel function, now taking 2 vectors and returning a vector of integers. You only have to modify the code in file mandel_VCL_avx.cpp.

Note that the most complex part of the function is already tackled, that is the part dealing with divergence checking and gathering of results. This had to be adapted, as all pixels of a given vector won’t necessarily run the same amount of iterations. Try to understand it and see how booleans are replaced by “masks” and how we continue looping until last pixel is over, while remembering the last loop number of each pixel already done.

Then complete the vectorization by adapting the few lines of the core computation, they are marked with comments “parts you have to work on”.

What did you have to change in practice ?

Once you have vectorized the kernel, compile the new version :

```
make libmandelVCL.so
```

and run via

```
python mandel.py VCL
```

How much time did the computation take in vector mode ?

What speedup did you achieve ?

Can you explain why the speedup is not perfect ?

6 Back to godbolt

Let’s now inspect out vectorized version in godbolt. Look at `https://godbolt.org/z/aczqjYoPW` and see how the kernel instructions are all packed (“ps” suffix), in particular on lines 30-50 of the assembly, where the computation takes place.