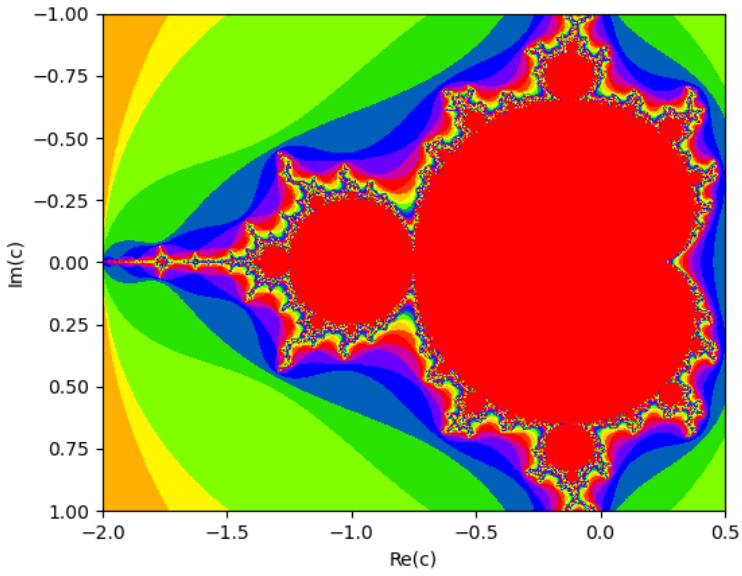
Exercise 5 : vectorization

Sébastien Ponce

October 7, 2025



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 'python3 mandel.py' will make use of libmandel.so while 'python3 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-2025-<nn> machine you were provided via ssh, with X support. This supposes you've already amended you .ssh/config file in the first exercise.

```
ssh -X <username>@csc-2025-<nn>
```

- 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 15 as a compiler

```
source /cvmfs/sft.cern.ch/lcg/releases/gcc/15.2.0-35657/x86_64-el9/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. Note that is the picture does not appear on the screen, you can check the mandelbrot_python.svg file saved locally.

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:.
python3 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/PeK7E1MGe, 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 23).

What proves that we did not get vector instructions ?

In the top of the right panel, click on "Add New" and add an "Opt Remarks" output. You can now see the code with comments inline concerning vectorization. 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', we are told that that loop was not vectorized. From the messages, can you extract the key point making autovectorization impossible ? It's not obvious, so do not hesitate to ask a teacher.

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

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

```
python3 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/jb37rYjrx` and see how the kernel instructions are all packed ("ps" suffix), in particular on lines 35-70 of the assembly, where the computation takes place.

7 Improving further

One of the key points we've learned from the courses is that we should ease the work of the compiler when it comes to low level optimization (pipelines, superscalar features) by having subsequent operations as independent as possible from each other.

The piece of code we are working on here is actually not respecting this rule at all. The inner loop (the 100 iterations) has almost every operation depending directly on the previous one. In particular the operations in the loop can hardly benefit from superscalar features as you need the previous iteration's result to compute the next one.

One way out would be to compute several items in parallel within the loop, that is have the kernel function taken an array of items instead of a single one. Note that I mention items here, quite a generic term, as this strategy applies both to scalar and vector cases. We'll try to use it on the vectorized code to see whether we can gain some more speed.

Pratically, we change again the main loop from

```
Vec8f ay{miny};
for (unsigned int any = 0; any < ny; ay += dy, any++) {
  Vec8f ax{minx, minx+dx, minx+2*dx, minx+3*dx,
    minx+4*dx, minx+5*dx, minx+6*dx, minx+7*dx};
  for (unsigned int anx = 0; anx < nx; ax += 8*dx, anx += 8) {
    kernel(ax, ay).store(buffer+any*nx+anx);
  }
}
```

to :

```
Vec8f ay{miny};
for (unsigned int any = 0; any < ny; ay += dy, any++) {
  // first N vectors of 8 pixels in x
  Array<Vec8f> ax{minx, dx};
  // loop 8*N by 8*N pixels in x
  for (unsigned int anx = 0; anx < nx; anx += N*8) {
    // compute and store back 8 pixels in one go
    auto res = kernel(ax, ay);
    res.store(buffer+any*nx+anx);
    ax += N*8*dx;
  }
}
```

So we call kernel on sets of N vectors (N = 2 in the provided code) and we use the `Array` type defined in `array.h` file. This implements an array of SIMD vectors with the same interface as the underlying vector type, and thus as the original scalar types. Each operation (`+`, `*`, `-`, `>`) is simply applied to all items one by one, hence having independent operations next to each other. There is also a specialization for arrays of booleans so that even the `horizontal_add` operation is provided. Have a look at `array.h` to have a better idea of how it works.

With this in mind, your new task is to complete the `mandel_Fast` implementation, and in particular the kernel method as you did previously for the VCL case. Again try you have to complete the part in between "parts you have to work on" comment but you should also try to understand the code already there.

> What did you have to change in practice ?

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

```
make libmandelFast.so
```

and run via

```
python3 mandel.py Fast
```

> How much time did the computation take for N = 2 ?

> What speedup did you achieve with N = 2 ?

This already is quite an achievement. But we should probably try higher Ns and see what is the optimal value. Fill the following table, changing the value of N in the source code and recompiling each time before you run

```
edit mandel_Fast_avx.cpp
make libmandelFast.so
python3 mandel.py Fast
```

N	2	3	4	5	6	7
Computation time						
Speedup up to scalar						

One can see that going too far is not optimal, probably because we do not manage anymore to fit all temporaries in registers. Anyway we got an extra gain :-)

8 Understanding the improvements via perf

Let's try to measure precisely how we got such a speedup in this last round. Our main tool will be the perf command and we will drop the python code and the graphical output so that we can measure more precisely the mandel method without overhead. For this purpose the file `main.cpp` is provided. It calls the mandel method the same way as in `mandel.py` with no extra processing.

We will compile 3 executables based on main.cpp, using the 3 implementations of mandel (scalar, VCL and Fast). They are called respectively `runMandel_scalar`, `runMandel_VCL` and `runMandel_Fast`.

Before you compile, make sure to set N back to the optimal value in `mandel_Fast_avx.cpp`

```
make runMandel
```

First run each executable individually and check that the processing time is still roughly the same as measured from python.

```
./runMandel_scalar
./runMandel_VCL
./runMandel_Fast
```

Now let's run each of them through the perf tool and look at statistics and in particular at the number of instruction per cycle ("insn per cycle" line)

```
perf stat ./runMandel_scalar
```

> Number of instructions per cycle for scalar case

```
perf stat ./runMandel_VCL
```

Number of instructions per cycle for vector case

```
perf stat ./runMandel_Fast
```

Number of instructions per cycle for fast case

See how the number of instructions per cycle went up thanks to superscalar features and puting together independent instructions.