Practical Work: Vectorization

Master CSMI
Compilation & Performance
Bérenger Bramas

November 4, 2020

1 Summary

In this current work, you will manually vectorize a code.

2 Ressources

- The Intel Intrinsics Guide: `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.
- Gcc documentation on the possible options: `https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html`.

3 Practical work organization (always the same)

In the practical work, you will obtain the code from my repository and push it to your repository. Therefore, you will have to clone one branch per session and push it to your own repository. It is mandatory that you **commit and push** frequently (after each question and at the end of the session, at least) such that I can easily look at what you have coded at the end of the session, how you did progress (and potentially compare it with the latest version you will have).

It is required that you filled the report.md file to let me know what you did.

In the rest of the document, we consider you have a repository named *cnp-tp-2020* on *git.unistra.fr* that is private but that I can access in read.

3.1 Get the practical work

Consider you are in your project directory do the following:

```
# Clone my repo
git clone https://git.unistra.fr/bbramas/csmi-tp-2020.git --branch=TP4 csmi-tp4
# If you use SSH, use:
# git clone git@git.unistra.fr:bbramas/csmi-tp-2020.git --branch=TP4 csmi-tp4
# Go in the newly created directory
cd csmi-tp4
```

3.2 Add your repository as remote

You will push on your own repository:

```
# Rename my remote
git remote rename origin old-origin
# Add your own remote
git remote add origin https://git.unistra.fr/[YOU LOGIN HERE]/cnp-tp-2020.git
# If you use an SSH key:
# git remote add origin git@git.unistra.fr:[YOU LOGIN HERE]/cnp-tp-2020.git
# Push the current branch and active the tracking
git push -u origin TP4
```

3.3 During the session and while you work on the project

After each question or important modification push the current changes:

```
# No matter where you are in the project directory
git commit -a -m "I did something"
git push
```

3.4 When you are done

You have fully finished your work (at most D+14 H-2):

```
git commit -a -m "I did something"
git push
```

3.5 Important!

Do not forget that you have to fill the report.md file and commit it. Remember to commit regularly to keep track of your work and let me see a history of it if I need it. Do not share any code with someone else, as I am here to answer all questions and support all of you. Remember that you have questions to answer in the moodle before the end of the session and that you must push your branch at the end of the session too. Additional credits can be obtained if you make some modifications after the session to improve your solution. Changes can be made until two weeks after the session minus two hours, ie you must push before the beginning of the n+2 practical work. Do not remove code from the test functions as I use them to evaluate your code. Therefore, if you need you can add extra functions for your own testing/debugging. If some of them are showing interesting things about your code, simply leave a comment in the code and the report.md.

3.6 Compilation

To compile, we use CMake:

```
cd TP4
mkdir build
cd build
cmake ..
make # Will make all
make something # Will build only something
VERBOSE=1 make # Will show the commands used to compile (including the flags)
```

4 Reminder

Vectorization is a capability of the CPU to execute a single instruction on multiple data (SIMD). On current modern hardware, it is implemented by having registers that can store several values, and instructions that can act on these registers. Therefore, this can also be seen as "vector" processing because the values we want to work on should be contiguous. Also, it is important to understand that there is a difference between the values in memory and the one in the registers.

Because not all CPUs support the same vectorization instruction sets, it is important to inform the compiler about the CPU we target. For instance, by default the compiler will optimize for a generic CPU (and will certainly not be able to vectorize). But, if we tell the compiler that we focus on a given hardware (and that the binary will not be executed on other/old systems) the compiler will start using specific instructions and might be able to vectorize.

One can provide specific options to turn on some of the CPU features, for instance *-mavx* to tell Gcc that it can use AVX instructions. Or, it is possible to speak in terms of hardware version, like *pentium2*, *atom*, etc. To do so Gcc have two flags:

- *-march=X*: allows GCC to generate code that may not run at all on processors other than the one indicated.
- *-mtune=X*: asks GCC to generate code that is better optimized for the process indicated (but remain compatible with what is specified by *-march*).

If one wants to optimize a binary for the CPU where the code is compiled, then *-march=native -mtune=native* should be used.

To facilitate the vectorization of codes, many compilers support "intrinsics". An "intrinsic" is a function that should translate into a single instruction by the compiler. Additionally, intrinsics use data-types that have the size of some CPU registers and thus should be directly mapped to registers and not allocated in the stack.

```
__m128d a, b; // __m128d is a data-type for 2 double real values
              // and which represent SSE registers

__m128d c = _mm_add_pd(a, b); // Will call an SSE instruction to some
                              // registers a and b, and store the result
                              // in a register c
```

5 Check the capability of the CPU

Make sure that your CPU support AVX using:

- *lscpu*
- *cat /proc/cpuinfo*

6 Does the compiler auto vectorize?

Using `https://godbolt.org/`, look at the asm of the *dot* function from the dot.cpp file. (use x86-64 gcc 9.1) You will see that it is not vectorized since all the instructions are for scalars and the floating point registers contain only one value (%xmmX).

To allow the compiler to use more instructions, we can use different options:

- *-mtune=native -march=native*: in our case this is not the best idea as it says to the compiler that the code should be optimize for the web server of godbolt.org.
- *-march=haswell -mtune=haswell*: Haswell processors support AVX2.
- *-mavx -mavx2*: be specific on the fact that Gcc can use AVX2 instructions but you might miss some features of the CPU you target.

However, as you will see, you will also need *-O3* to have the compiler doing auto-vectorization.

Go back on your terminal, and compile while looking at the options (*VERBOSE=1 make*), and ensure to have all the flags you need to have auto vectorization.

7 AVX2 dot kernel

In dot.cpp, you will find the function *dot_sse3* that perform the dot using SSE3 instructions. Use it as a model to implement the dot in AVX2 in the empty function *dot_avx2* (remember that AVX2 is AVX plus extra things). In the first version consider that the memory is not aligned, and then implement a second version considering that the vector are 64 bytes aligned.

To help you, the function *hsum* that perform the horizontal sum (the sum of all the elements of a vector) is provided. This function should be use at the end.

Here is an possible performance difference that you should obtain (notice that the process is pinned to a core):

```
$ taskset -c 0 ./dot
...
idx = 50000
>> Scalar timer : 0.0147024
>> SSE3 timer : 0.0110316
>> AVX2 timer : 0.0058162
>> AVX2 aligned timer : 0.00564722
```

On my PC, vectorizing by hand provides an important speedup over auto-vecotorize code (and this difference can be even more significant for complexe codes).

8 4x4 matrix/matrix product

Create an AVX2 kernel in the matmat.cpp file to compute the 4x4 matrix product. As you will see in *matmat4x4* we consider that we use the transpose of matrix *B*. The function *h4sum* is provided: this function does the horizontal sum of 4 vectors and stores the result in a vector. Remark: As you can see in the code, the matrix are allocated in the stack (no call to new/malloc) but are aligned using the C++ *alignas(32)* keyword, and thus can safely be loaded into register using the *load* for aligned memory instruction.

```
$ taskset -c 0 ./matmat
Check matmat4x4
>> Scalar timer : 0.00981972
>> AVX2 timer : 0.00518744
```