ACCELERATOR BASED PROGRAMMING
UPPSALA UNIVERSITY
FALL 2023

ASSIGNMENT 1: USING THE CPU AND THE GPU

Due: September 15, 2023 Upload a report on Studium.

In this assignment, we will get started with programming for single instruction multiple-data (SIMD) on classical CPU architectures as well assome basic CUDA programming on the GPU.

Please also consult the guide on our Studium page.

1. Provided program files. The experiments in this lab are based on the following C++ programs available from the course page on Studium:

- `stream_triad.cc`: Basic C++ implementation of the STREAM triad benchmark in single precision.
- `stream_triad_simd.cc`: C++ implementation of the STREAM triad benchmark with a SIMD abstraction class around intrinsics.
- `vectorization.h`: Wrapper class around intrinsics on x86-64 hardware needed for the stream triad simd.cc class.
- `stream_triad_cuda.cu`: CUDA/C++ implementation of the STREAM triad benchmark in single precision

2. Necessary preparation. The SIMD part relies on the content presented in the `01_CPUarch_SIMD.pdf.pdf` file of the lecture. Read the relevant parts of the notes to get acquainted with the concepts used in this assignment. The first step is to log in on UPPMAX with your user name. In case you do not have an account yet, please have a look at the instructions given in `00_CourseIntroduction.pdf`. Then, proceed with the following steps

- Log in to the machine with
  `ssh USERNAME@rackham.uppmax.uu.se`
  on Linux and OSX and using an ssh application form Windows (UU recommends MobaXterm).
- To request one CPU node on Snowy for an interactive session, use the command
  `salloc -A uppmax2022-2-19 -N 1 -M snowy -t 1:0:00`
  You can compile code, run programs, and do basic experiments.
- To request one GPU node on Snowy for an interactive session, use the command.
  `salloc -A uppmax2022-2-19 -N 1 -M snowy --gres=gpu:1 --gpus-per-node=1 -t 1:0:00`
- Read the instructions on UPPMAX for further details, e.g. for using a project file.
- To compile CPU (host) code with the C++ compiler, use the compiler `g++` (basic version) or, by using
  `module load gcc/11.2.0`,
  for a newer version.
- To compile GPU code with CUDA, we need to use the `nvcc` compiler. To use it, we need to load the modules by the two commands
  `module use /sw/EasyBuild/snowy/modules/all/`
  `module load intelcuda/2019b`

3. Benchmark description. For this assignment, we work primarily with the STREAM triad benchmark. It runs the following code on a vector of size N

```
for (int i = 0; i < N; ++i)
    z[i] = a * x[i] + y[i];
```

Here, `x,y,z` are vectors (array pointers) in single precision (float), whereas `a` is a scalar This benchmark is limited by the available memory bandwidth with decent code, so the typical metric to assess the performance is GB/s (gigabytes

per second), computed by recording the time t of the experiment and then computing

$$BW = 10^{-9} \frac{3[\text{vectors}] \times N[\text{vectors}] \times [\text{bytes/float}]}{t} \tag{3.1}$$

Due to the overhead of timers for small sizes, we repeat the benchmark several times, record the time for the overall experiment, and include the number of repetitions in the numerator of the above formula. Furthermore, like all computer experiments, our benchmark is subject to some noise. We therefore repeat the experiment 20 times and record the best, average, and minimal time for each of these repetitions.

4. Tasks.

1. Familiarize yourself with the code in stream triad.cc. Then, compile the code with two different levels of optimization:
   (a) `g++ -march=sandybridge -O2 stream_triad.cc -o stream_triad`
   (b) `g++ -march=sandybridge -O3 stream_triad.cc -o stream_triad`

2. For the former case, the compiler will generate scalar code, whereas the latter will result in vectorized (SIMD) code. Run the experiments for $N = 8, \ldots, 10^8$, and plot the achieved memory bandwidth for both cased with the command:
   `./stream_triad -min 8 -max 1e8`
   Try to explain the differences you see.

3. If you have the possibility to run on another hardware than UPPMAX, you will likely use a more powerful single-core processor. Compile the code with the `-march=native -O3` option instead and record the achieved memory performance.

4. Read the code in `vectorization.h` and compile the code with the option
   `g++ -march=sandybridge -O2 stream_triad_simd.cc -o stream_triad`
   and record the performance again. Again, try to use the native processor on a newer architecture as well. You can select between two versions, one with aligned memory using the **run time option** `-align 1` and one without alignment `-align 0` (default). is this just work with x86?

5. Compile the CUDA version of the code with
   `nvcc stream_triad_cuda.cu -o stream_triad_cuda`
   and run it on a GPU node with similar arguments as above. Compare the throughput on the CPU and the GPU. What about the performance for small and large sizes, respectively? smaller size CPU is slightly better than GPU but large size GPU is significantly better than CPU should I also run from 8 to 1e8 or 1e4 to 1e8?

6. The CUDA code involves a parameter const int block_size = 512; defined at the top of the file. Modify the size between 1 and 2048 and discuss the achieved performance

7. Port the GPU program from single precision to double precision. Record the performance again and discuss the metrics MUPD/s (million updates per second) and GB/s between single and double precision. What is the factor limiting performance?