TECHNIQUES AND TECHNOLOGIES FOR SCIENTIFIC SOFTWARE ENGINEERING FTN0439
UPPSALA UNIVERSITY
SPRING 2023

LAB 1: USING THE CPU AND THE GPU

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

Please also consult the USING UPPMAX 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 first 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 have not yet an account, please have a look at the instructions. 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 uppmax2023-2-20 -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 uppmax2023-2-20 -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`
- If we are many people in the lab we should not log into an interactive session on each node but use the job scripts provided on studium. We have priority for the afternoon of our lab so we get immediate allocation if we use our reservation code.

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 cases with the command:
   `./stream_triad -min 8 -max 1e8`
   Try to explain the differences you see.

3. 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?

4. The CUDA code involves a parameter `const int block_size = 512;` defined at the top of the file. Modify the size between 1 and 1024 (Max blocksize) and discuss the achieved performance. We will discuss this parameter tomorrow.

5. 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?