

Lab 3 - Introduction to SIMD programming

Oguz Kaya
oguz.kaya@universite-paris-saclay.fr

You can consult the Intel's webpage for AVX functions as well as their latencies for each architecture:

<http://software.intel.com/sites/landingpage/IntrinsicsGuide>

It would be sufficient to filter for AVX, AVX2 and FMA type instructions on the leftside of the page when searching for an instruction.

To compile the program `program.cpp` with AVX/AVX2 instruction sets and generate the executable `program`, type the following command in the terminal:

```
g++ -O1 -std=c++11 -mavx2 -mfma program.cpp -o program
```

Part 1

Copying an array using SIMD

The goal of this exercise is to learn the basics of SIMD programming using AVX, by copying an array into another array using vectorized load/store instructions.

Ex. 1

- Allocate two float arrays `A` and `B` of size N , then initialize `A` such that `A[i] = i`. Make sure that the array allocation is aligned by 32 bytes, in order to be able to use aligned load/store instructions (which are typically faster).
- Implement a non-vectorized function that copies the content of `A` to `B`.
- Implement a vectorized version that performs the same operation.
- Implement a third version that unrolls (by hand) the loop by a factor of 4, and carries out 4 iterations of the previous version in a single “big” iteration.
- Compare the runtime of each version for 1000 subsequent executions for $N = 1024$. Does the unrolled version go faster, and why/why not?

Part 2

Vector inner product using SIMD

The goal of this exercise is to compute the inner product of two vectors x and y :

$$x^T y = \sum_{i=1}^N x_i y_i$$

using vectorization.

Ex. 2

- Allocate two float arrays `x` and `y` of size N (divisible by 8), then initialize (`x[i] = i` and `y[i] = 1` for instance).
- Write a scalar (non-vectorized) function that computes the inner product of `x` and `y`.
- Write a second version that performs the same operation, but with vectorized load/store/add instructions.
- Implement a third version that unrolls the loop by a factor of 2 and 4. How many cycles is your code expected to spend per iteration i) without unrolling, ii) unrolling of 2, iii) unrolling of 4? You can check the instruction latencies
- Modify the multiplications and additions with fused-multiply-add (FMA) operations. Does it go faster?

- f) Compare the execution time of all version for 1000 consecutive executions.
- g) Try to automatically vectorize the scalar version simply by adding the -ftree-vectorise flag to the compilation, then re-test the execution time. Did it accelerate?

Part 3

Reversing an array using SIMD

The goal of this exercise is to reverse an array of N floats using SIMD instructions. To facilitate, you can assume that N is always a multiple of 8.

Ex. 3

- a) Write a function that reverses an array of floats. To do this, use the AVX function `__mm256_permutevar8x32_ps(__m256 a, __m256i idx)` to revert a vector of 8 floats, then iterate over both sides of the array to swap vectors of 8 floats on each side after reverse-permuting them.

Part 4

Matrix-vector product using SIMD

The goal of this exercise is to perform matrix-vector multiplication using SIMD programming.

Ex. 4

- a) Write a program that performs the multiplication of a matrix $A \in \mathbb{R}^{N \times N}$ with a vector $x \in \mathbb{R}^N$ and store it in another vector $y = Ax, y \in \mathbb{R}^N$. The matrix-vector multiplication is as follows:

$$y_i = \sum_{j=1}^N A_{ij} x_j$$

You can store the matrix row-wise., and suppose that N is a factor of 8 for simplicity.