TP - Introduction to SIMD programming

Oguz Kaya, Atte Torri, Matthieu Robeyns
{oguz.kaya,atte.torri,matthieu.robeyns}@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++ -O2 -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

a) 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 alinged load/store instructions (which are typically faster).

b) Implement a non-vectorized function that copies the content of `A` to `B`.

c) Implement a vectorized version that performs the same operation.

d) 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.

e) Compare the runtime of each version for 1000 subsequent executions for $N = 1024$. Does the unrolled version go faster, and why/why not? (You may look at load and store bandwidth for your CPU model on wikichip).

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

a) Allocate two float arrays `x` and `y` of size N (divisible by 8), then initialize (`x[i] = i` and `y[i] = 1` for instance).

b) Write a scalar (non-vectorized) function that computes the inner product of `x` and `y`.

c) Write a second version that performs the same operation, but with vectorized load/store/add instructions.

d) 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

e) 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. You can reuse your OpenMP code if you have it, and add SIMD parallelization on top of of it.

Ex. 4

a) Écrire un programme qui effectue le produit d'une matrice $A \in \mathbb{R}^{N \times N}$ avec un vecteur $x \in \mathbb{R}^N$ dans un autre vecteur $y = Ax, y \in \mathbb{R}^N$. Le calcul s'effectue comme la suite:

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

Vous pouvez aussi prendre le code OpenMP non-vectorisé du dernier TP, puis le vectoriser. Y a-t-il une amélioration des performances? Pour quelles valeurs N est-il le cas?

b) 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.