Exercises for High Performance Computing (MA-INF 1108) WS 2024/2025

U. Sinha, C. Penke
Tutors: N. Pillath, M. I. A. Khan

4 Vectorization

1: Vector addition

The goal of this exercise is to understand the performance improvements achieved with different levels of vectorization for a compute kernel that performs addition of two arrays

$$c[i] = a[i] + b[i]$$

a) [2pt] Complete the TODOs in the source code `vector_addition.c` by implementing the functions `vector_add_plain` for floating point arrays and `vector_add_plain_double` for double arrays.

b) [3pt] Compile your function with (`gcc`) using increasing optimization degree (`-O0, -O1, -O2, -O3`). Save the data containing the `serial no., size,` of the array, and `time` taken to complete the vector additions for both floating point and double arrays in a `txt` file in the `solutions` folder. Use `objdump` to disassemble the different executable files. Compare the assembly codes. Reply the following questions related to the function `vector_add_plain` and `vector_add_plain_double` in a `a041b.txt` file in the solutions folder.

   - Which compiler optimization levels use scalar instructions?
   - Which compiler optimization levels use packed vector instructions?
   - Which compiler optimization levels use 32-bit registers?
   - Which compiler optimization levels use 128-bit registers?
   - Which compiler optimization levels use 256-bit registers?

Note: To obtain measurable time for the different sizes of the arrays, take the size of the arrays at powers if 2 multiplied with $1,000,000$. For e.g. $1000000, 2000000, 4000000$.....

c) [1pt] Generate the vectorization report for all the optimization levels and save it in a `txt` file in a separate folder named `vectorization_report`. The vectorization report can be generated by using the flag `-fopt-info-vec-all` (an example for optimization level `-O3` is

```
gcc -O3 -lm
-fopt-info-vec-all=vectorization_report/vector_addition_O3.txt
src/vector_addition.c -o bin/vector_addition_O3
```
).

2: Vector addition with AVX2 intrinsics

a) [2pt] Complete the TODOs in the source code `vector_addition_avx2.c` by implementing the functions `vector_add_avx2` for floating point arrays and `vector_add_avx2_double` for double arrays.

b) [3pt] Compile your function with (`gcc`) using increasing optimization degree (`-O0, -O1, -O2, -O3`). Save the data containing the `serial no., size,` of the array, and `time` taken to complete the vector additions for both floating point and double arrays in a `txt` file in the `solutions` folder. Use `objdump` to disassemble the different executable files. Compare the assembly codes. Reply the following questions related to the function `vector_add_avx2` and `vector_add_avx2_double` in a `a042b.txt` file in the solutions folder.

- Which compiler optimization levels use scalar instructions?
- Which compiler optimization levels use packed vector instructions?
- Which compiler optimization levels use 32-bit registers?
- Which compiler optimization levels use 128-bit registers?
- Which compiler optimization levels use 256-bit registers?

Note: To obtain measurable time for the different sizes of the arrays, take the size of the arrays at powers if 2 multiplied with $1,000,000$. For e.g. $1000000, 2000000, 4000000$.....

c) [1pt] Generate the vectorization report for all the optimization levels and save it in a `txt` file in a separate folder named `vectorization_report`. The vectorization report can be generated by using the flag `-fopt-info-vec-all`.

3: Matrix Multiplication

The goal of this exercise is to understand the performance improvements achieved with different levels of vectorization for a compute kernel that does a matrix multiplication

$$C = A \cdot B \Rightarrow C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

where A is an $m \times n$ and B is an $n \times p$ matrix, and the product $C = A \cdot B$ will be an $m \times p$ matrix, where each element is C_{ij} with i and j are the row and column indices of the matrix C

a) [2pt] Complete the TODOs in the source code `matmul.c` by implementing the `matrix_multiply_plain` function.

b) [3pt] Compile your function with (`gcc`) using increasing optimization degree (`-O0, -O1, -O2, -O3`). Save the data containing the `serial no., size,` and `time` taken to complete the matrix multiplication for a given matrix size in a `txt` file in the `solutions` folder. Use `objdump` to disassemble the different executable files. Compare the assembly codes. Reply the following questions related to the function `matrix_multiply_plain` in a `a043b.txt` file in the solutions folder.

c) [1pt] Generate the vectorization report for all the optimization levels and save it in a `txt` file in a separate folder named `vectorization_report`. The vectorization report can be generated by using the flag `-fopt-info-vec-all` (an example for optimization level `-O3` is

```
gcc -O3 -lm -fopt-info-vec-all=vectorization_report/matmul_O3.txt
src/matmul.c -o bin/matmul_O3 ).
```

4: Matrix Multiplication with AVX2 intrinsics

a) [2pt] Complete the TODOs in the source code `matmul_avx2.c` by implementing the `matrix_multiply_avx2` function.

b) [3pt] Compile your function with (`gcc`) using increasing optimization degree (`-O0, -O1, -O2, -O3`) together with the `-mavx2` flag. Save the data containing the `serial no., size,` and `time` taken to complete the matrix multiplication for a given matrix size in a `txt` file in the `solutions` folder. Use `objdump` to disassemble the different executable files. Compare the assembly codes. Reply the following questions related to the function `matrix_multiply_plain` in a `a044b.txt` file in the solutions folder.

c) [1pt] Generate the vectorization report for all the optimization levels and save it in a `txt` file in a separate folder named `vectorization_report`. The vectorization report can be generated by using the flag `-fopt-info-vec-all` (an example for optimization level `-O3` is `gcc -O3 -lm -mavx2 -march=native -fopt-info-vec-all=vectorization_report/matmul_avx2_O3.txt src/matmul_avx2.c -o bin/matmul_avx2_O3`).

5: Plotting the performance results

Visit the link `https://gitlab.jsc.fz-juelich.de/sinha3/python-plotting-for-hpc-course` to check the plotting scripts discussed in lecture 2. Using those scripts, answer the following questions:

a) [2pt] Write a python plotting script to plot the data generated by `vector_addition.c`. The plot should describe the vector addition time versus the size of the vector for both the floating point and double arrays and the different optimization levels. Describe the results and your conclusion in caption.

b) [2pt] Write a python plotting script to plot the data generated by `vector_addition_avx2.c`. The plot should describe the vector addition time versus the size of the vector for both the floating point and double arrays and the different optimization levels. Describe the results and your conclusion in caption.

c) [4pt] Write separate python plotting scripts to plot the data generated by `matmul.c` and `matmul_avx2.c` and plot the time versus matrix size. Create another plot comparing the results of matrix multiplication with and without avx intrinsics and write your conclusions in the caption..