DUE DATE: Before the end of your lab session

CSE-443/543: High Performance Computing
Exercise #10

Max Points: 20

You should save/rename this document using the naming convention MUid.docx (example: `raodm.docx`).

Objective: The objective of this exercise is to review the use of SSE/AVX instructions for vectorization of loops

1. Explore automatic vectorization of loops in `g++`
2. Understand the need to use `-ffast-math` in some situations.

Fill in answers to all of the questions. You may discuss the questions with your instructor.

Name: Maciej Wozniak

Background: Streaming SIMD Extensions (SSE)

In computing, Streaming SIMD Extensions (SSE) is a Single Instruction Multiple Data (SIMD) extension that is often used to accelerate common array or vector (mathematical concept) operations. A key enhancement of SSE over SIMD is that SSE permits data to be perfected while the previous data is being processed to enable effective operations on a list of values.

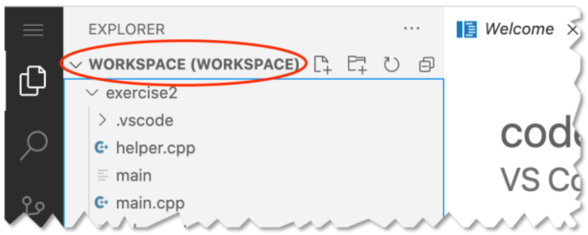
Advanced Vector Extensions (AVX) further extend SSE instructions with wider registers (or more operands) with sophisticated FMA (fused multiply-accumulate, i.e. 1 instruction for `a[] += b[] * c[]`, were `a`, `b`, and `c` are arrays) instructions that are widely used in graphics, multimedia, and cryptographic processing.

Exploring automatic vectorization

Download and save the supplied `exercise8.cpp` program to your work folder. This program is a standard C++ program that performs basic mathematical operations on a list of single-precision floating-point numbers (i.e., `float` data type). In this exercise, we will be using some of GCC's flags (i.e., command-line arguments) to see the effect of vectorization.

Task 1: Project setup steps:

Estimated time: 5 minutes



1. Log into OSC's OnDemand portal via https://ondemand.osc.edu/. Login with your OSC id and password.
2. Startup a VS-Code server and connect to VS-Code. Ensure you switch to your workspace. Your VS-Code window should appear as shown in the adjacent screenshot.

3. Next, create a new VS-Code project in the following manner:
    a. Start a new terminal in VS-Code
    b. In the VS-Code terminal use the following commands:

```
$ # First change to your workspace directory
$ cd ~/cse443
$ # Use ls to check if workspace.code-workspace file is in pwd
$ # Next copy the basic template for a C++ project
$ cp -r /fs/ess/PMIU0184/cse443/templates/basic exercise10
$ # Copy the starter code for this exercise
$ cp /fs/ess/PMIU0184/cse443/exercises/exercise10/* exercise10
```

4. Add the newly created project to your VS-Code workspace to ease studying the source code.

Task #2: Study the supplied source code
Estimated time: 10 minutes

Study the supplied `exercise10.cpp`. It is a very straightforward C++ program and you should be able to easily explain the following operations of this program:

1. What is SAXPY? What does the acronym SAXPY stand for?

    Single-Precision A.X + Ym where Z =A*x+Y

2. Why is SAXPY targeted by hardware manufacturers (both CPU & GPU) and compiler developers?

    It is because it is highly vectorizable which is good for GPUs (since they are focus on matrix operations) and CPUs since it will test SSE optimization

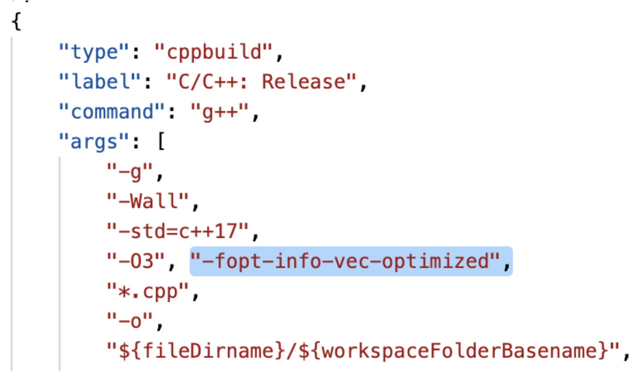
Task #3: Explore automatic vectorization
Estimated time: 5 minutes

Background: The GCC compiler (similar to several other C++ compilers) has the ability to automatically vectorize parts of a C++ program to take advantage of SSE instructions to improve performance (without sacrificing energy efficiency). For enabling and using automatic vectorization, the following two GCC flags (i.e., command-line arguments) will be used:

1. `-O3` : This optimization level automatically engages GCC's vectorization optimizations. The vectorizer analyzes the source code and vectorizes parts that it recognizes (so if you code it too cleverly the vectorizer may not work well with your code).
2. `-fopt-info-vec-optimized`: We will add this flag to ask GCC to print information about parts of the program it actually managed to vectorize. Without the flag GCC will still vectorize the code at `-O3` optimization level but just wont print messages.

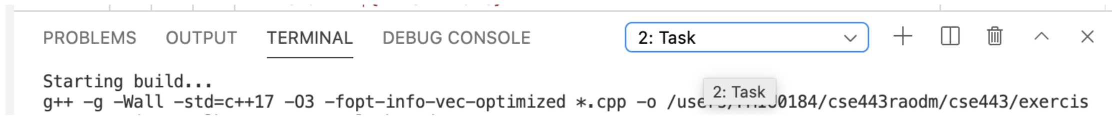
Exercise:

1. In your projects, `tasks.json`, add the compiler flag `-fopt-info-vec-optimized` to the release build as shown in the screenshot below:



```
{
    "type": "cppbuild",
    "label": "C/C++: Release",
    "command": "g++",
    "args": [
        "-g",
        "-Wall",
        "-std=c++17",
        "-O3", "-fopt-info-vec-optimized",
        "*.cpp",
        "-o",
        "${fileDirname}/${workspaceFolderBasename}",
```

2. Next run the project. You should observe outputs (`note` messages) from GCC indicating that parts of the program are being vectorized in the Task terminal window shown in the screenshot below:



```
Starting build...
g++ -g -Wall -std=c++17 -O3 -fopt-info-vec-optimized *.cpp -o /use      0184/cse443raodm/cse443/exercis
```

   Copy-paste the messages from GCC in the text box below:

```
/apps/gnu/8.4.0-offload-nvptx/include/c++/8.4.0/bits/stl_vector.h:296:18: note: basic
block vectorized
exercise10.cpp:40:24: note: basic block vectorized
exercise10.cpp:55:19: note: loop vectorized
exercise10.cpp:55:19: note: loop vectorized
exercise10.cpp:26:26: note: loop vectorized
exercise10.cpp:26:26: note: loop versioned for vectorization because of possible aliasing
```

   You should notice the following types of messages. Some messages will be repeated due to 2-pass nature of optimization (first-pass tags and second-pass checks and optimizes):
   a. Message stating the loop was vectorized.
   b. Loop was "peeled" -- that is parts of the loop was repeated to handle sizes that are not multiples of SSE sizes or parts that do not align well with caches
   c. Versioned loops -- if you have two references the compiler cannot say for sure if they are pointing to the same vector or to 2 different vectors and needs to handle that at runtime using 2 different versions of machine code.

3. Using the output from the compiler answer the following questions:
   a. Which two methods in the source code were successfully vectorized:

      `Saxpy, forloop in makeList, sum,`

   b. Which methods in the source code were not vectorized:

      `Main ? it seems that all of them were vectorized`

Task 4: Potential impediments to full vectorization

Background: In the previous part of the exercise, depending on the compiler, you could notice that the `sum` method may not be vectorized to perform many additions in parallel using multiple ALUs and SSE instructions. Ideally, we would like to have the `sum` method also to be vectorized, for example: `x[i] + x[i + 1]` in 1 ALU and `x[i + 2] + x[i + 3]` in another ALU, where `x` is a list of numbers and `i` is a valid index into the array.

So, if the compiler should be able to do it, then why don't the compilers always do it? The answer to that question requires us to go back to CSE-174 and trace the output of the following straightforward program (no, there is nothing tricky here):

```
#include <iostream>

int main() {
    const float a = 1.1, b = 2.2, c = 3.3;
    const float d = (a + b) + c;
    const float e = a + (b + c);
    std::cout << (d - e) << std::endl;
    return 0;
}
```

Trace the operation of the above program. What is the expected output from the program?	0

Now run the above short program. To make life easier, you can just copy-paste the main method to your starter code and run it.

What is the actual output from the program?	4.76837e-07

Inference: You should notice a discrepancy in expected output (which should be zero) versus observed output because floating-point arithmetic in computers is not associative -- that is, the standard concepts of mathematics do not hold true! The actual output is highly sensitive to the order of operations. If a million numbers (a tiny size of data in big data analytics context) are added the error can be as high as ±1 -- or as hyperbole a space shuttle could land-up in Jupiter instead of Mars! Sure higher precision numbers (such as: `long double` at 16 bytes) reduce the error and that is why scientists clamor for more precision in floating point arithmetic all the time leading to 512-bit floating point numbers.

However, vectorization breaks order of operations to have multiple operations run in parallel. Consequently, the compiler generally favors preserving order of operations (to yield consistent results to serial order of addition) and does not vectorize the loop.

Task 4: Understanding relaxed or fast-math

Background: Knowing that floating-point arithmetic is approximate in computers is very important, particularly for engineering, scientific applications, big-data analytics, statistics, etc. The compiler developers also know it and they have done their best to preserve order of operations so that a deterministic program (exact same inputs, exact same instructions) will yield deterministic amount of errors that can be accounted for.

However, what happens if an application is willing to accept some differences between each run of a deterministic program to improve performance? Sure that is possible and that is where fast-math mode in C++ compilers comes into the picture (you can also do fast-math in Java using 3rd party libraries) as detailed at: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Specifically, **fast-math** refers to a family of optimizations, namely: `-fno-math-errno`, `-funsafe-math-optimizations`, `-ffinite-math-only`, `-fno-rounding-math`, `-fno-signaling-nans`, `-fcx-limited-range` and `-fexcess-precision=fast`.

Exercise:
In this part of the exercise, we will engage fast-math mode to relax requirements and vectorize the `sum` method in the program using the following procedure:

1. Using the procedure from Task #3, add `-ffast-math` flag to the C++ compiler's options.
2. Next compile-and-run the project. You should observe outputs (`note` messages) from GCC indicating that parts of the program are being vectorized. Copy-paste the messages from GCC in the text box below:

```
g++ -g -Wall -std=c++17 -O3 -fopt-info-vec-optimized -ffast-math *.cpp -o
/users/PMIU0184/wozniamk/CSE543_wozniak/exercise10/exercise10 -lboost_system -lpthread
exercise10.cpp:26:26: note: loop vectorized
exercise10.cpp:26:26: note: loop versioned for vectorization because of possible
aliasing
exercise10.cpp:40:24: note: loop vectorized
/apps/gnu/8.4.0-offload-nvptx/include/c++/8.4.0/bits/stl_vector.h:296:18: note: basic
block vectorized
exercise10.cpp:40:24: note: basic block vectorized
exercise10.cpp:55:19: note: loop vectorized
```

From the above output you should notice that the loop in the `sum` method has now been vectorized, if it wasn't being vectorized earlier.

Task 5: Summary from exercise

In the space below summarize the key learning outcomes from this exercise by briefly (1 or 2 sentences each) answering the questions below:

1. What is vectorization? What is the name for the 2 types of instructions used for vectorizing a program?

   Loop was vectorize -> we used vector extension to run program faster. It used sse instruction to vectorized the program
   SSE, SIMD

2. What parts of the program (*i.e.*, language constructs) are/were vectorized?

   Mainly loops (part of functions)

3. What optimization flag enables automatic vectorization in GCC?

   -O3

4. What happens if the `-fopt-info-vec-optimized` option is omitted?

   Without the flag GCC will still vectorize the code at `-O3` optimization level but just wont print messages.

5. What is key issue that could hinder vectorization of some methods in this exercise?

   Versioned loops -- if you have two references the compiler cannot say for sure if they are pointing to the same vector or to 2 different vectors and needs to handle that at runtime using 2 different versions of machine code.

6. What is the work around that is built into C++ compilers to work around the above issue and enable vectorization?

   Adding `-ffast-math flag for compilation`

Submit files to Canvas

Upload the following files to Canvas:

1. Upload this MS-Word document (duly filled with the necessary information) saved as PDF using the naming convention **`MUid.pdf`**.