Vectorization using C/C++ x86-64 instrinsics

Objectives.

1. To enable gcc compiler to auto-vectorize C/C++ code
2. To write C programs using Intel SSE intrinsics.
3. To write C programs using Intel AVX intrinsics

Aim

The aim of this tutorial is to learn how to optimize C/C++ applications using x86-64 vectorization intrinsics

Introduction

All modern processors support vectorization. This means that processors have extra hardware components (wide registers and wide processing units) to allow vector processing. Vectorization is the process of processing vectors (multiple values together) instead of single values. Vectorization is also known as Single Instruction Multiple Data (SIMD), as a single instruction is used to process multiple data.

Vectorization dramatically improves the performance of our code. The compilers apply vectorization automatically, this process is called auto-vectorization, but not always with success. Vectorization has nothing to do with the language used.

Vectorization can be applied in four main ways by

1. automatically by the compiler (specify '*-O3*' option or '*-ftree-slp-vectorize*' '*-ftree-vectorize*'). Compile using '*-fopt-info-vec-optimized*' option to see which parts are vectorized. This solution does not provide the best performance, but it is very easy to use.
2. using OpenMP C/C++ pragmas. Better performance than above, less easy to use (this will be studied later on).
3. using C/C++ instrinsics (assembly coded functions). Better performance than above, not that easy to use.
4. directly writing assembly code (not used in this module). Best performance but hard to use.

By using C/C++ instrinsics we can get high performance (comparable to assembly - assembly code is always faster than any other language) portable and somehow easy to write code.

The code for this week's lab session is located to the 'Vectorization 1st lab session' folder. Download '*code to start*' source files.

For those who use Visual Studio: Open Visual studio and create an empty (C/C++) project. Copy paste the code provided in your project. **For more information about how to create a C/C++ project please see the '*How to Create a C/C++ Project*'** *section* below. Visual Studio does not support a separate template for C. Note that C++ is a superset of C. So, C++ supports everything that comes with C plus extra features such as Object-Oriented Programming, Exception Handling and a rich C++ Library. So, we will create a C++

project and write C code. We could rename all the .cpp files to .c files and run strictly C code, but there is no reason for doing that.

The code provided, contains four .cpp and three .h source files. It is good practice to put all the declarations in the .h files and the definitions to the .c/.cpp files. The execution of the program always starts from the main() function which is located to the main.cpp file.

Section1 – Auto-vectorization

Download the '*code to start'* folder from Vectorization/section3/ directory. Compile the code using one of the following options:

gcc main.cpp array_addition.cpp array_constant_addition.cpp MVM.cpp -o p -march=native -O2 -fopt-info-vec-optimized -ftree-vectorize

gcc main.cpp array_addition.cpp array_constant_addition.cpp MVM.cpp -o p -march=native -O3 -fopt-info-vec-optimized

-fopt-info-vec-optimized' option to see which parts are vectorized. Run and measure performance

Section 2 – Using omp #pragmas (the easy way)

This section is just to give you an idea of how easy vectorization becomes with OpenMP. We will be focusing on OpenMP during the next weeks. OpenMP provides a set of compiler directives that are used to provide extra information to a compiler to allow it to automatically parallelise and/or vectorise code (typically loops). These are built into the compiler and accessed by using pragmas (via #pragma). Pragmas are hints that the compiler can choose to use or ignore, depending on whether it has built-in support for that capability. OpenMP 4.0 introduced omp simd, accessed via #pragma omp simd as a standard set of hints that can be given to a compiler to encourage it to auto-vectorise code.

A few examples are shown below

```
#pragma omp simd
For (int n=0; n<N; ++n)
  a[n] += b[n];   //the compiler will vectorize this if possible
```

```
#pragma omp simd aligned(a,b:16) safelen(4)
for (int i=0; i<N; ++i)
  a[i] = a[i-4] * 2;
```

Safelen tells the compiler that it is safe to vectorise 4 values at most. Aligned tells the compiler that a and b arrays are 16byte aligned.

There is a large number of clauses supported by OpenMP such as reduction, simdlen etc, to allow efficient and easy vectorization even in cases where the loop kernel is complex. OpenMP will not be studied this week.

Section 3 - using C/C++ instrinsics (input size is multiple of 4/8 only)

I would recommend the remaining tasks to be performed in Visual Studio as it provides better support when dealing with larger codes. Alternatively, you can use Eclipse in Linux or even a simple text editor and terminal.

1st example, add an array with a constant number

This is the 1st code example which adds an array with a constant number and stores the result to another array. This example is explained in the slides too. The files involved to this example are just the *main.cpp*, the *array_constant_addition.cpp* and the *array_constant_addition.h*. *array_constant_addition.h and array_constant_addition.cpp* contain three different implementations of the aforementioned algorithm : a) a normal C routine, b) a routine using SSE instrinsics, c) a routine using AVX instrinsics. You can specify which function to run in main.cpp file under the 't' loop.

In main.cpp file there is code initializes the arrays (initialization_ConstAdd() function), code used to measure the execution time of 't' loop, code for verifying that the ConstAdd_SSE() and ConstAdd_AVX() generate the right output etc.

Task1. **Build and run the code.**

By default, the Const_Add_SSE() routine will run and a message will be printed depending on whether the vectorized version of the algorithm generates the same output as the non-vectorized one. All the functions defined in *array_constant_addition.cpp file, return the same output, which is the value of 2. This way, main() knows that we need to compare the output of this algorithm and thus the appropriate if condition will be executed. Inside the if condition there are two commands. The first one stores a message into an array of strings using snprintf(). The second, prints a message whether the output is correct or not. To do so, the Compare_ConstAdd() routine is executed.*

Task2. Measure the execution time

In this task, we will measure the execution time of *Const_Add_SSE()* routine. **Make you sure you run the project using Ctr+F5 (run without debug);** debugging mode does not provide the actual execution time of the program. **It is important to note that for an accurate measurement, the execution time needs to be at least some seconds.** Thus, given that the execution time of the current routine is lower than 1sec at all times, we run it *'t'* times ('*t*' is a loop variable) and then the overall execution time is divided to the number of iterations. The upper bound of *'t'* is the 'TIMES_TO_RUN' macro and must be appropriately defined.

Comment the *Const_Add_SSE()* routine and uncomment the *Const_Add_default()* routine. Build and run the program and measure the execution time of the *Const_Add_default()* routine. This routine must be slower. Repeat the experiment and measure the execution time of *Const_Add_AVX()* routine. This routine is faster than *Const_Add_SSE(), as 8 elements are processed in parallel, not 4.*

Task3. Study the *Const_Add_SSE() and Const_Add_AVX()* routines

Make sure you understand how the *Const_Add_SSE() and Const_Add_AVX()* routines work. *Const_Add_SSE() is explained in the slided. Const_Add_AVX() does the same thing but used AVX technology and thus 8 elements are processed at once. For now, assume that the input size is a multiple of 8. Next week, we will further explain the general solution. All the C instrinisics* can be found in https://software.intel.com/sites/landingpage/IntrinsicsGuide/# .

Drawing upon the slides, try to understand how the routines work.

__declspec(align(64)) : Do not forget to initialize the arrays using this command. When programming in SSE/AVX all the arrays must be defined as '***__declspec(align(64)) float array[N];*** ' . This command directs the compiler to align the arrays to a 64-byte boundary. The starting address of the array is '(*starting_address modulo 64 = 0*)'. By doing so, the first element of the array (*array[0])* is always stored into a first cache line slot.

num2 = _mm_loadu_ps (&v2[i]) : This command loads 4 v2[] elements from memory and stores them to the 128bit variable num2. These elements are v2[i], v2[i+1], v2[i+2], v2[i+3]. Each array element is of 32bits, so their sum is 128 bits. Keep in mind that num2 is defined as ***' __m128 num2*** '. The input operand of this command must be a memory address (you can check here https://software.intel.com/sites/landingpage/IntrinsicsGuide/#); therefore the ***&*** operator must be used so as the memory address of ***v2[i]*** to be provided.

num2 = _mm_load_ps (&v2[i]) : This command is similar to ***_mm_loadu_ps().*** However, the ***&v2[i]*** **must be aligned to a 16-byte (or 128-bit) boundary**. This means that this instruction works for v2[0], v2[4], v2[8], v2[12], v2[16] etc inputs only and not v2[1], v2[2], v2[3]. This instruction loads 4 elements (128bits), but the starting element must a multiple of 4 or equivalently a memory address aligned to a 128bit boundary. So, ***_mm_load_ps (&v2[1])*** will give an error.

_mm_store_ps (&v1[i], num3) : This command is similar to ***_mm_load_ps().*** However, a store is performed and not a load. The contents of the 128bit variable num3 are stored into the memory address ***&v1[i]*** , and therefore the values of v1[i], v1[i+1], v1[i+2] and v1[i+3] are updated.

num3 = _mm_add_ps (num1, num2) : This command will add the packed 4 32bit values of num1 to the packed 4 32-bit values of num2 and put the result into num3.

The instructions above use the SSE technology and process 128bits of data. AVX technology use similar instructions to SSE but processes 256-bit of data. The 256bit registers are defined as ***__m256 ymmm***; . ***_mm_add_ps*** becomes ***_mm256_add_ps*** and ***_mm_load_ps*** becomes ***_mm256_load_ps.***

2nd example, arrays addition

This is the 2nd code example which adds two arrays and stores the result to another. The files involved to this example are the *array_addition.cpp* and the *array_addition.h*. *array_addition.h and array_addition.cpp* contain three different declarations of the aforementioned algorithm : a) a normal C routine, b) a routine using SSE instrinsics, c) a routine using AVX instrinsics. The routines (b) and (c) must be written by you. You can specify which function to run in main.cpp file under the 't' loop.

Task1. Implement the Add_SSE() and Add_AVX() routines: Drawing upon Section1, you will implement the Add_SSE() and Add_AVX() routines. These routines are very similar to those in section1.

Task2. *Measure the execution time:* Measure and compare the execution time of the three routines. The default input size is very small, so you will need to increase it, like in array_constant_addition.h.

3rd example, Matrix-Vector multiplication

This is the 3rd code example which multiplies a matrix by a vector (see slides for more information). The files involved to this example are the MVM.*cpp* and the MVM.*h*. *MVM.h and MVM.cpp* contain three different implementations of the aforementioned algorithm : a) a normal C routine, b) a routine using SSE instrinsics, c) a routine using AVX instrinsics. You can specify which function to run in main.cpp file under the 't' loop.

Task1. Study the MVM_SSE() routine

Drawing upon the slides, try to understand how the MVM_SSE() routine works. The MVM_AVX() routine is more complicated and it is only presented for consistency (you do not have to study it).

num3 = _mm_fmadd_ps (num0, num1, num3) : This command will multiply the packed 4 32bit values of num0 by the packed 4 32-bit values of num1 and then add the 4 32bit results to the packed 4 32bit values of num3. So, this command applies both multiplication and addition. This command can be broken down into two instructions (one multiply and one add), but this would increase the execution time.

_mm_store_ss (&Y[i], num4) : This command stores just the lower 32bit value from num4 to the memory address of &Y[i].

xmm1 = _mm_hadd_ps (xmm0, xmm1) : This command horizontally adds adjacent pairs of 32-bit values in xmm0 and xmm1 and pack the results into xmm1 (Fig.1).

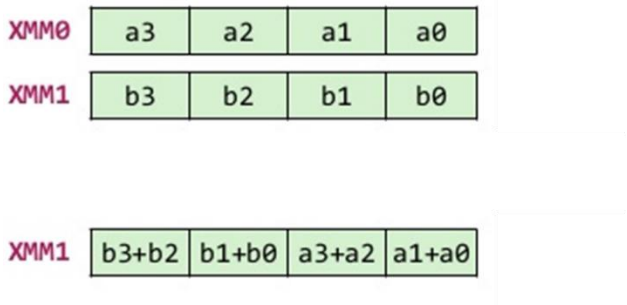

Fig.1 _mm_hadd_ps() command. The registers/variables contain 4 32bit values.

Task2. Measure the execution time: Measure and compare the execution time of the three routines.

Section 4 - using C/C++ instrinsics (general case – any input size)

The implementations we have studied so far refer to input sizes that are multiples of 4 only. This is because in each iteration, four elements were loaded and processed (the iterator is increased by a factor of 4). Thus, if the input size is 6, then only the first 4 iterations will be processed, leaving the last two iterations unprocessed. In this task, we will extend the MVM_SSE() routine to all the input sizes.

Task1. Study the new MVM_SSE() routine provided.

Two changes have been made to the previous MVM_SSE() routine. Firstly, we needed to amend the upper bound of j loop. This must always be a multiple of 4, otherwise the code in the loop body does not work properly. For example, if the input size is 6, the upper bound must be 4, if the input size is 9, the upper bound must be 8. The rest iterations will be processed next without using vectorization. This is implemented as follows:

*Upper_j_bound=(M/4) *4;*

The M/4 division is between integers, and thus in C, the result of the division will be an integer (the lowest integer value). Thus, 6/4 will give 1 instead of 1.5, 10/4 will give 2 instead of 2.5, etc. Thus, the above formula, always gives the number we are looking for. For example, if M=6 then *Upper_j_bound=4, if M=10 then Upper_j_bound=8, if M=16 then Upper_j_bound=16.*

Secondly, a padding code was added in the end of j loop to execute the remaining iterations. For example, if M=10, then Upper_j_bound=8 and thus only the first 8 iterations are vectorized; the last two iterations must be processed normally. The starting value of j is missing, which means that j will have the last j value used; alternatively, we could specify the starting j value in the padding code as j=(M/4)*4 (but this is not needed).

Section 5 – More Advanced Examples

Task1. Implement MMM_SSE() routine

This is the 2nd code example which mutiplies two 2d arrays and stores the result to another. The files involved to this example are the *MMM.cpp* and the *MMM.h*. *MMM.h and MMM.cpp* contain three different declarations of the aforementioned algorithm : a) a normal C routine, b) a routine using SSE instrinsics, c) a routine using AVX instrinsics. The routine (b) must be written by you. You can specify which function to run in main.cpp file under the 't' loop.

Drawing upon MVM_SSE(), you will implement the MMM_SSE() routine. These two routines are very similar with each other. However, you will not succeed if you do not draw on the paper the three matrices and understand how the three loop iterators work (Fig.1).

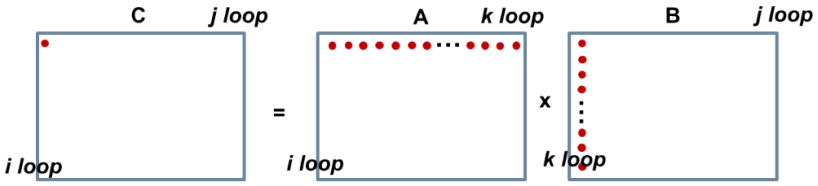

Fig.1. A visual representation of MMM algorithm

The B[][] array is accessed in a column-wise order. Each row of A[][] is multiplied by a column of B[][]. However, the elements of each column are not written in consecutive memory locations. Thus, the load instruction we have seen so far cannot load B[0][0], B[1][0], B[2][0] and B[3][0] array elements. Note that mm_load_ps(&B[0][0]) will load B[0][0:3] not B[0:3][0].

To address the above issue, we create the B[][] transpose and store it into a new array (Btranspose[][]). The first row of Btranspose is the first column of B, the second row of Btranspose is the second column of B etc. This is implemented as follows

```
for (i=0; i< N; i++)
for (j=0; j< N; j++)
Btranspose[i][j] = B[j][i];
```

Thus, the new loop kernel will be the following

```
for (i=0; i< N; i++)
for (j=0; j< N; j++)
for (k=0; k< N; k++)
C[i][j] += A[i][k] * Btranspose[j][k];
```

*This way, a row of A[][] is multiplied by a row of Btranspose[][] and the above problem is solved. N**ow you are ready to implement the MMM_SSE() routine,** drawing upon the MVM_SSE().*

Task2. Learn how to vectorize if-conditions.

For those who need to learn a bit more about vectorization, I have included an example vectorizing an if-conditions code. This task is optional. The files involved to this example are the if_cond.*cpp* and the if_cond.*h*. The way this code works is explained in the the slides. All the C instrinsics are found here https://software.intel.com/sites/landingpage/IntrinsicsGuide/#

Further reading

1. *Virtual Workshop (Cornell University), available at https://cvw.cac.cornell.edu/vector/overview_simd*
2. *Tutorial from Virginia University, available at https://www.cs.virginia.edu/~cr4bd/3330/F2018/simdref.html*

How to Create a C/C++ Project

Visual Studio does not support a separate template for C. **Please note that C++ is a superset of C**. So, C++ supports everything that comes with C plus extra features such as Object-Oriented Programming, Exception Handling and a rich C++ Library. **So, we will create a C++ project and write C code**.

Open Visual studio and select 'create new project' as in Fig.2. Then, type 'C++' in the search area in the top and select the 'empty project' option (Fig.3). Specify the project's name and directory and click 'create' button (Fig.4). Right click on the 'source files' and select 'add'-> 'new item' (Fig.5). Then select 'C++ File (.cpp)' and click 'Add' button (Fig.6). Name the .cpp file appropriately and copy paste the code provided for this file. Repeat this for the other .cpp files too. For the header files (.h), right click on the 'header files' and select 'add'-> 'new item'. Then select 'header File (.h)' and click 'Add' button. Name the .h file appropriately and copy paste the code provided for this file. Repeat this for the other .h files too.

If you want to run strictly C code, then rename all the .cpp files to .c files.

Press F7 to build the code (this option is under the Build tab). Then, press Ctrl+F5 to run the program without debugging or press F5 to run with debugging. To measure the actual execution time of the program you must run the code without debugging (Ctrl+F5).

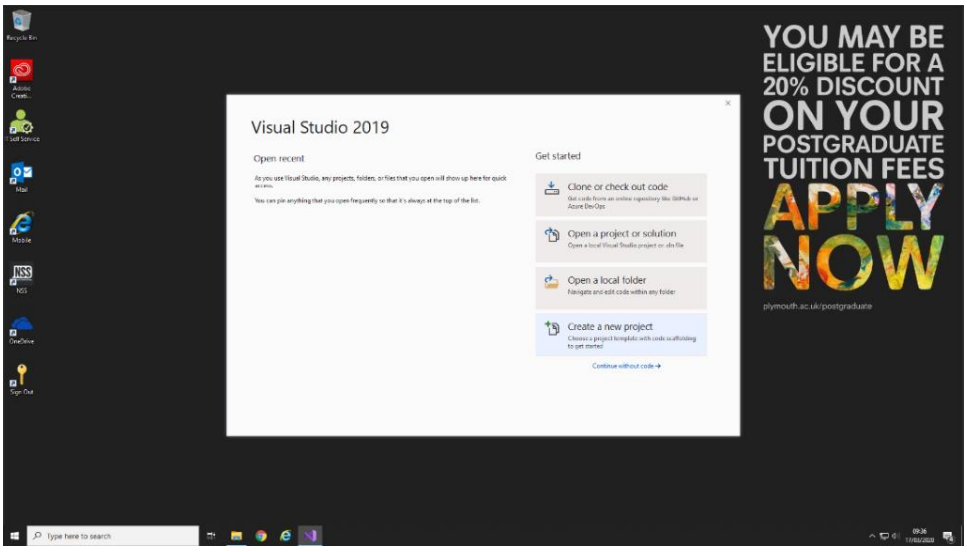

Fig.2. How to create a C++ Project Step1

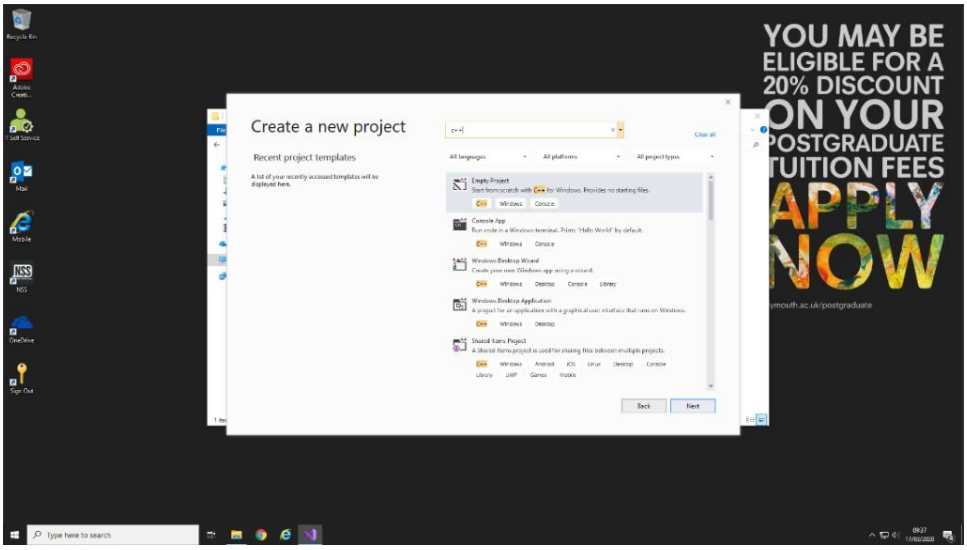

Fig.3. How to create a C++ Project Step2

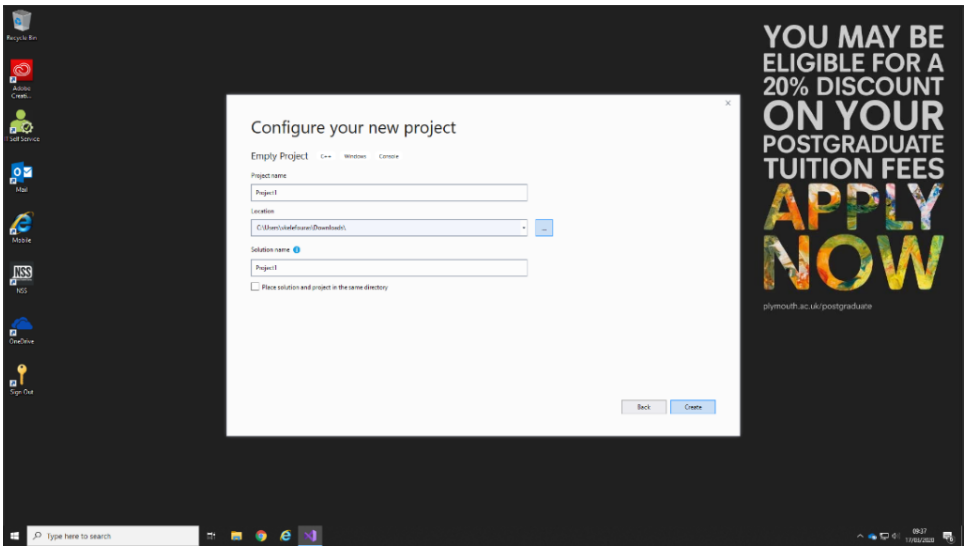

Fig.4. How to create a C++ Project Step3

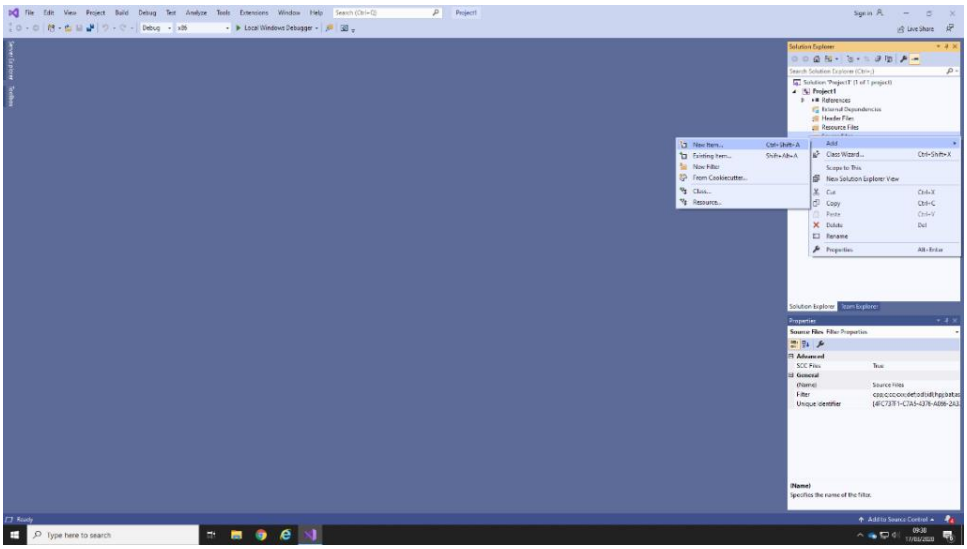

Fig.5. How to create a C++ Project Step4

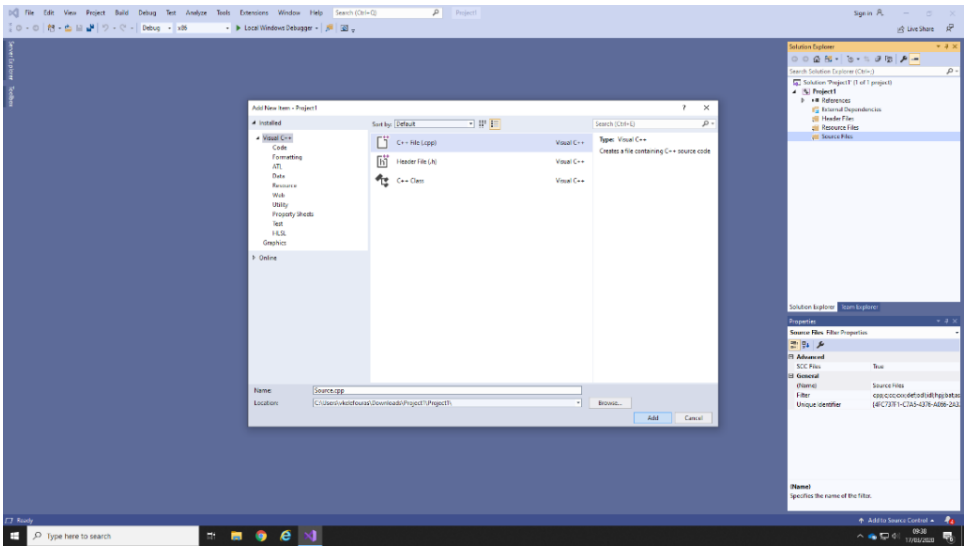

Fig.6. How to create a C++ Project Step5