Vectorization 2nd Session - Using x86-64 intrinsic functions in C/C++

Objectives.

1. To understand C/C++ programs using Intel SSE/AVX intrinsics.
2. To write simple C/C++ programs using Intel SSE/AVX intrinsics given a code template

Aim

Writing assembly code can be hard, so another option is to write assembly in C/C++ using intrinsicics (assembly coded functions). The aim of this session is to learn how to write simple C programs using SSE/AVX x86-64 instrinsics given a code template.

Introduction

This is the 2nd and last lab session for vectorization. The code skeleton is very similar to the one used in the previous lab session. This lab session builds upon the previous one, so make sure you have completed the previous session.

There are three source files and two header files. The main source file is main.cpp, which calls either the arrays constant addition example we have seen previous week, or the arrays addition example which we will study at this tutorial. Every example has its header file where all the declarations are included and its source file where all the definitions are included.

Section 1 – Arrays addition example

This example 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) must be written by you. You must specify which function to run in main.cpp file under the 't' loop. The 'array_constant_addition.cpp' and 'array_constant_addition.h' files contain the previous week's example.

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.

Task1. Implement the Add_SSE() routine

Study the 'Add_default()' routine in 'Array_Addition.cpp' file. Drawing upon the previous tutorial, you will vectorize this routine using SSE intrinsics (you will write your code in the Add_SSE() routines). If you want to stretch your programming skills try to implement this routine by using AVX instrinsics too. You will use exactly the same intrinsics as those you used previous week, shown in ConstAdd_SSE() and ConstAdd_AVX() routines in 'array_constant_addition.cpp' file. In the previous tutorial, we have seen an example where an array is added with a constant number. This week we will add an array with another array.

If you are confused about how these intrinsics work, you can put a breakpoint inside a vectorised routine and then click debug. In the autos and local windows you will see the wide registers and their values. Press F11 to go step by step and check the registers' values.

See below an explanation of the intrinsics to be used.

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] and v2[i+3]. Each array element is 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. Remember that '&' means memory address of.

_mm_storeu_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.

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

Task2. *Measure the execution time*

Measure and compare the execution time of the three routines. To get an accurate execution time measurement, the program must run for at least a few seconds. So, you might need to amend the 'TIMES' variable in main(). Which routine runs faster?

Task3. Amend all the SSE routines to run successfully for any input size.

The implementations we have studied so far refer to input sizes that are multiples of 4 only. 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 SSE() routines to support all different input sizes.

Two changes must be made to the SSE() routines.

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 of the 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 must be 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 code is shown below. 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).

The code follows. The new parts are shown in red.

```
unsigned short int ConstAdd_SSE() {

__m128 num1, num2, num3;
int i;

num1 = _mm_set_ps(2.1234f, 2.1234f, 2.1234f, 2.1234f); //set num1 values

for (i = 0; i < ((M / 4) * 4); i += 4) {//e.g., if M==10, then ((10/4)*4)=8 as the
division is between integers
	num2 = _mm_loadu_ps(&V2[i]); //load 4 elements of V2[]
	num3 = _mm_add_ps(num1, num2); //num3 = num1 + num2
	_mm_storeu_ps(&V1[i], num3); //store num3 to Y[i]. num3 has 4 FP values which
they are stored into Y[i], Y[i+1], Y[i+2], Y[i+3], respectively
	}

//padding code
for (; i < M; i++) { //equivalently you could write ' for (j=(M/4)*4; j < M; j++) '
	V1[i] = V2[i] + 2.1234f;
}

return 2;
}
```

Section 2 – Matrix-Vector multiplication

This example will be discussed in the revision session. Before you start working on this example revise the previous examples and make sure you understand how they work. This is a code that multiplies a matrix by a vector. You can find the code for this lab session on Github, in the '***Vectorization 2nd lab session***' directory.

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_default() routine

Drawing upon the 'MVM_default()' routine and Fig.1, try to understand what this program does.

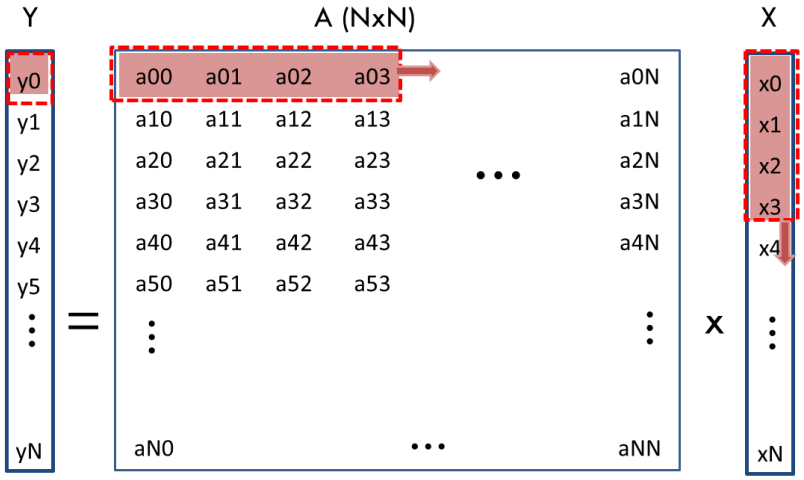

Fig.1 An Illustration of MVM program.

Task2. *Measure the execution time*

Measure and compare the execution time of the three routines.

Task3. Study the new MVM_SSE() routine provided.

Study the 'MVM_SSE()' routine in 'MVM.cpp' file. Fig.2 further explains the procedure.

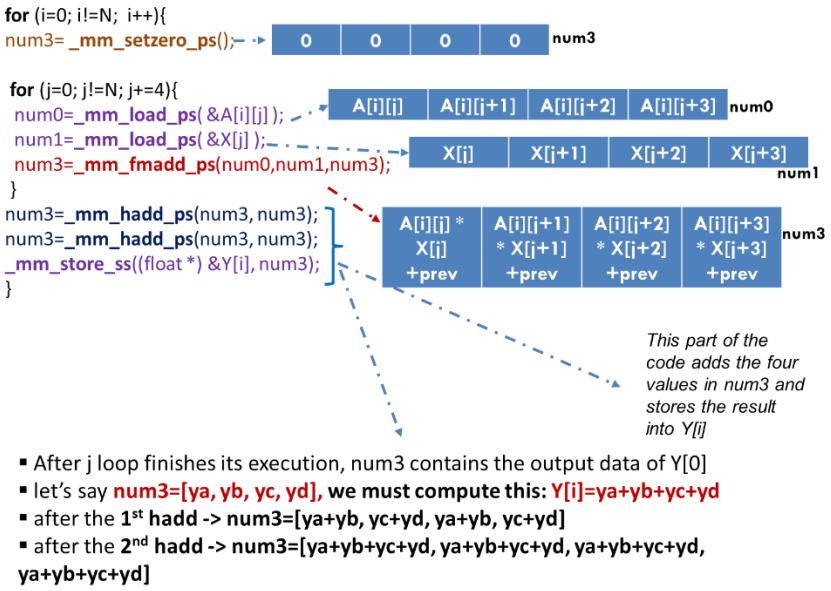

Fig.2 MVM_SSE routine explained

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.

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).

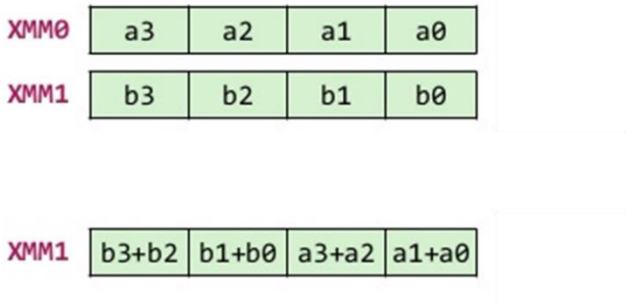

Fig.1 _mm_hadd_ps() command. The registers/variables contain 4 32bit values.

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.***

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*