Vectorization and Parallelization of Loops in C/C++ Code

Xuejun Liang, Ali A. Humos, and Tzusheng Pei
Department of Computer Science, Jackson State University, Jackson, MS, USA

Abstract - *Modern computer processors can support parallel execution of a program by using their multicores. Computers can also support vector operations by using their extended SIMD instructions. To make a computer program run faster, the time-consuming loop computations in the program can often be parallelized and vectorized to utilize the capacity of multicores and extended SIMD instructions. In this paper, the vector multiplication and the matrix multiplication will be used as examples to illustrate how to perform parallelization and vectorization of loops in a C/C++ program when using Microsoft Visual C++ compiler or GNU gcc (g++) compiler. An overview of the Intel@ Advanced Vector Extension (AVX) instructions, their intrinsics, and the OpenMP is given. The performance testing results and their comparisons are also presented for the combinations of cases, such as with or without vectorization or parallelization.*

Keywords: Computer Architecture, Vectorization, Advanced Vector Extension (AVE), Parallelization, OpenMP

1 Introduction

Modern computer processors can support parallel execution of a program by using their multicores. Computers can also support vector operations by using their extended SIMD instructions. To make a program run faster, the time-consuming loop computations in the program can often be parallelized and vectorized to utilize the capacity of multicores and extended SIMD instructions.

There are two ways to vectorize a loop computation in a C/C++ program. Programmers can use intrinsics inside the C/C++ source code to tell compilers to generate specific SIMD instructions so as to vectorize the loop computation. Or, compilers may be setup to vectorize the loop computation automatically. This is called auto-vectorization. There are also two ways to parallelize a loop computation in a C/C++ program. Programmers can use pragmas (like those defined in OpenMP) inside the C/C++ source code to guide compilers to parallelize the loop computation. Or, compilers may be setup to parallelize the loop computation automatically. This is called auto-parallelization.

In this paper, the vector multiplication as shown in Figure 1 and the matrix multiplication as shown in Figure 2 and 3 are used as computation examples. The Microsoft Visual C++ compiler and the GNU gcc (g++) compiler are investigated and utilized to compile the example programs. This paper will illustrate how to use intrinsics and pragmas to perform the loop vectorization and the loop parallelization, respectively. It will also show how to setup compiler flags to perform auto-vectorization and auto-parallelization. An overview of the Intel@ Advanced Vector Extension (AVX) instructions, their intrinsics, and the OpenMP is given. The performance testing results and their comparisons are also presented for the combinations of cases with or without the two types of vectorization or the two types of parallelization. These testing results and comparisons will tell us that in what situation programmers should use intrinsics and pragmas explicitly to perform the vectorization and the parallelization, respectively, or otherwise, let the compiler do the job. They also show that memory access patterns will affect the performance.

```
for(int i = 0; i < MAX_DIM; ++i)  //A
  c[i] = a[i] * b[i];
```

Figure 1: Vector Multiplication

```
for(int i = 0; i < n; ++i) {      //B1
  for(int j = 0; j < n; ++j){     //B2
    c[i*n+j] = 0;
    for(int k = 0; k < n; k++) {  //B3
      c[i*n+j] += a[i*n+k] * b[k*n+j];
    }
  }
```

Figure 2: Matrix Multiplication (stored in Row-Major)

```
for(int i = 0; i < n; ++i) {      //C1
  for(int j = 0; j < n; ++j){     //C2
    c[i+n*j] = 0;
    for(int k = 0; k < n; k++) {  //C3
      c[i+n*j] += a[i+n*k] * b[k+n*j];
    }
  }
```

Figure 3: Matrix Multiplication (Stored in Column-Major)

Note that matrices are stored in one-dimensional arrays in the row-major order in Figure 2, and in the column-major order in Figure 3.

In the rest of the paper, an overview of the Intel@ Advanced Vector Extension (AVX) and their intrinsics is introduced in Sections 2. Vectorizing matrix multiplication by using AVX instrinsics is presented in Sections 3. The compiler options for auto-vectorization are given in Section 4. The compiler options for auto-parallelization and the OpenMP pragmas for loop parallelization are given in Section 5. The performance analysis and comparison are reported in Section 6. The conclusion is given in Section 7.

2 AVX Instructions and Intrinsics

The concept of SIMD (single instruction Multiple Data) is to apply a single operation on multiple data simultaneously. The computer capacity can be effectively enhanced by adding such SIMD instructions along with providing additional registers which can hold multiple scalar data.

Intel has experienced SIMD extensions many times from Multi-Media Extension (MMX), Streaming SIMD Extension (SSE, SSE2, SSE3, SSE4), to Advanced Vector Extension (AVX, AVX2, AVX-512). The AVX floating point registers extend from (XXM) 128-bit to (YYM) 256-bit, which can hold eight 32-bit single precision floating point (FP) operands or four 64-bit double precision FP operands.

Intel® AVX added support for many new instructions and extended Intel SSE instructions to the new 256-bit registers, by putting prefix "v" to SSE instructions for accessing new register sizes and three-operand forms [1]. Many instructions have also suffixes [PS/PD/SS/SD], where PS means packed single-precision, PD means packed double-precision, SS means scalar single-precision, and SD means scalar double-precision. Therefore, AVX could have the following forms of floating point addition instructions.

```
ADD[PS/PD/SS/SD]   XMM1, XMM2/M128
VADD[PS/PD/SS/SD]  XMM1, XMM2, XMM3/M128
VADD[PS/PD/SS/SD]  YMM1, YMM2, YMM3/M256
```

where M128 is memory data aligned at 128-bit boundary and M256 aligned at 256-bit boundary.

In order to use these AVX instructions in a C/C++ program, users will use intrinsics (intrinsic functions) that the compiler can replace with the proper assembly instructions. The Intel AVX intrinsic functions use three new C data types:

```
__m256, __m256d, and __m256i
```

where __m256 is packed float, __m256d is packed double, and __m256i is packed integer. Most Intel AVX intrinsic names have the following format:

```
_m256_OP_SURFIX
```

where `OP` is an operation such as `ADD` or `SUB`, and `SURFFIX` can be `[PS/PD/SS/SD]`. Three examples of prototypes of AVX intrinsic functions corresponding to the above three forms of AVX instructions are listed below.

```
__m128 _mm_add_ps(__m128 m1, __m128 m2);
__m128 _mm256_add_ps(__m128 m2, __m128 m3);
__m256 _mm256_add_ps(__m256 m2, __m256 m3);
```

Note that the prototypes for Intel AVX intrinsics are available in the header file immintrin.h.

3 Vectorizing Using AVX Intrinsics

Single-precision floating point numbers are used in this work. So, it is expected that 8 float numbers will be packed into one YYM register. Figure 4 shows the code of vectorized matrix multiplication using AVX intrinsics where matrixes are stored in the row-major order. Note that in Figure 4, we have

```
m0 = (c[i*n+j+0], ..., c[i*n+j+7])
m1 = (a[i*n+k+0], ..., a[i*n+k+0])
m2 = (b[k*n+j+0], ..., b[k*n+j+7])
```

```
for (int i = 0; i < n; i++) {          //D1
  for (int j = 0; j < n; j += 8) {     //D2
    __m256 m0 = _mm256_setzero_ps();
    for (int k = 0; k < n; k++) {      //D3
      __m256 m1 = _mm256_broadcast_ss(a+i*n+k);
      __m256 m2 = _mm256_load_ps((b+k*n+j));
      __m256 m3 = _mm256_mul_ps(m1, m2);
      m0 = _mm256_add_ps(m0, m3);
    }
    _mm256_store_ps(c+i*n+j, m0);
  }
}
```

Figure 4: Vectorized Matrix Multiplication Using Intrinsics (Matrixes Stored in Row-Major)

Figure 5 shows the code of vectorized matrix multiplication using AVX intrinsics where matrixes are stored in the column-major order. Note that in Figure 5, we have

```
m0 = (c[i+n*j+0], ..., c[i+n*j+7])
m1 = (a[i+n*k+0], ..., a[i+n*k+7])
m2 = (b[k+n*j+0], ..., b[k+n*j+0])
```

```
for ( int i = 0; i < n; i += 8 ) {     //E1
  for ( int j = 0; j < n; j++ ) {      //E2
    __m256 m0 = _mm256_setzero_ps();
    for( int k = 0; k < n; k++ ) {     //E3
      __m256 m1 = _mm256_load_ps(a+i+n*k);
      __m256 m2 = _mm256_broadcast_ss(b+k+n*j);
      __m256 m3 = _mm256_mul_ps(m1, m2);
      m0 = _mm256_add_ps(m0, m3);
   }
    _mm256_store_ps(c+i+l*j, m0);
  }
}
```

Figure 5: Vectorized Matrix Multiplication Using Intrinsics (Matrixes Stored in Column-Major)

4 Auto-Vectorization

Compiler options for automatically generating vectorized loop are presented for Microsoft Visual C++ and GNU GCC in this section.

4.1. Microsoft Visual C++ Compiler: Microsoft Visual Studio Express 2013 for Windows Desktop is used for this work. To start, create a new empty Visual C++ project. Then, select Release as the solution configuration and x64 as the solution platform. Finally, create a C++ file and add it to the project under Source Files folder. Now, we are ready to compile and run the program with Visual C++.

By default, Visual C++ will perform auto-vectorization in the release configuration. In order to get the reason code [2] to find out why a loop is not vectorized, we need to open the project's Property pages by clicking PROJECT and then Properties (Alt+F7), and select, in this order, Configuration Properties, C/C++, Command Line, and then write

```
/Qvec-report:2
```

in the textbox below Additional Options in the right pane. In addition, we can select Code Generation below C/C++ and choose

```
Advanced Vector Extensions (/arch:AVX)
```

by clicking on the right side of the line "Enable Enhanced Instruction Set" in the right pane to specify a targeted SIMD extension of IA32 for the vectorization.

Now, we can test the auto-vectorizer with code segments as shown in Figure 1, 2, and 3. The loop A in Figure 1 is vectorized only. Loops B1and B2 in Figure 2 and C1and C2 in Figure 3 are not vectorized because they are outer loops. Loops B3 in Figure 2 and C3 in Figure 3 are not vectorized because the loop body includes non-contiguous accesses into an array.

In order to disable auto-vectorizing a loop, we need to add the following line just before the loop statement.

```
#pragma loop(no_vector)
```

4.2. GCC Compiler: Minimal GNU for Windows (MinGW) will provide a native Windows port of the GNU Compiler Collection (GCC). GCC (gcc and g++) 4.6.2 and 6.1.0 are installed on the Microsoft Windows 7 platform and used for this work.

The compiler flag

```
-ftree-vectorize
```

will enable the auto-vectorization. This flag is enable at

```
-O3
```

Meanwhile, a targeted SIMD extension to IA-32, say, AVX, must be specified by using the compiler flag

```
-mavx
```

Thus, the following command will vectorize loops in the program, mytest.cpp

```
g++ -O3 -mavx mytest.cpp
```

In order to know if a loop is not vectorized and why, the flag

```
-ftree-vectorizer-verbose=2
```

should also be used.

Now, we can test the auto-vectorization with code segments as shown in Figure 1, 2, and 3. The loop A in Figure 1 and the loop B2 in Figure 2 (outer loop) are vectorized. The loops B1 and B3 in Figure 2 and C1 in Figure 3 are not reported. The loop C2 in Figure 3 is not vectorized because of complicated access pattern. The loop C3 in Figure 3 is not vectorized because of unsupported use in statement.

5 Auto-Parallelization and OpenMP

Compiler options for automatically generating parallelized loop are presented for Microsoft Visual C++ and GNU GCC in this section. Meanwhile, using OpenMP to parallelize a loop computation is also introduced.

5.1. Microsoft Visual C++ Compiler: To enable auto-parallelizer for parallelizing loops, we need to select Code Generation below C/C++ and choose

```
Yes(/Qpar)
```

by clicking on the right side of the line "Enable Parallel Code Generation" in the right pane. In order to get the reason code to find out why a loop is not parallelized, we need to select Command Line below C/C++ and then add

```
/Qpar-report:2
```

in the textbox below Additional Options in the right pane.

Now, we can test the auto-parallelizer with code segments as shown in Figure 1, 2, and 3. No loops are parallelized!

The loop A in Figure 1 is not parallelized because the compiler detected that this loop does not perform enough work to warrant auto-parallelization. In this case, you can tell the compiler to do the loop parallelization anyway by adding the following line just before the loop statement.

```
#pragma loop(hint_parallel(4))
```

Loops B1 and B2 in Figure 2 and C1 and C2 in Figure 3 are not parallelized because a data dependency in the loop body is falsely detected by the compiler. The loops B3 in Figure 2 and C3 in Figure 3 are not parallelized because there is a scalar reduction in the loop body. Scalar reduction can occur if the loop has been vectorized.

5.2. GCC Compiler: The following two compiler flags

```
-floop-parallelize-all
-ftree-parallelize-loops=4
```

will trigger the auto-parallelization. Therefore, the following command will parallelize loops in your program, mytest.cpp.

```
g++      -floop-parallelize-all      -ftree-
parallelize-loops=4 mytest.cpp
```

Unfortunately, g++ 4.6.2 does not implement Graphite loop optimization, which is supposed to perform the loop auto-parallelization. The latest version of g++ (6.1.0) do compile successfully with the two parallelization flags. But, no report is available to tell which loop is parallelized and which not.

5.3. Parallelize Loop Using OpenMP: OpenMP is an application programming interface (API) that supports shared memory multiprocessing programming in C, C++, and Fortran. Microsoft Visual C++ compiler supports OpenMP by default. Users only need to include the OpenMP header file omp.h in their program. GCC compiler supports OpenMP by using the compiler flag

```
-fopenmp
```

To parallelize a loop in C/C++ program, only a single OpenMP compiler pragma (directive)

```
#pragma omp parallel for
```

is needed to put right before the loop statement. Programmers can call the OpemMP function

```
omp_set_num_threads(NUM_THREADS)
```

to specify the number of threads to use.

6 Performance Analysis and Comparison

First of all, the Dell Precision M4600 laptop is used for the testing. The processor (Intel(R) Core i7-2620M CPU) has 2 cores and supports 4 threads. The auto-parallelization is tested only on the vector multiplication example with Visual C++. The run-time with auto-parallelization is around 10 times slower than without parallelization. As shown in Table 1, the run-time of auto-vectorization alone is very slow and program compiled by g++ is much faster than Visual C++ compiler. Note that four threads are used and the size of the arrays is 81920 for VC++ and 163840 for G++.

Table 1: The run-times of vector multiplication

	auto-vector	AVX	OpenMP auto-vector	OpenMP AVX
VC++	0.000050	0.000005	0.000005	0.000005
G++	0.000067	0.000001	0.000001	0.000001

As shown in Table 2, the run-times are almost the same with 4 threads and 2 threads. The run-times are also almost the same with and without OpenMP for VC++. Note that the size of matrices is 256×256 for VC++ and 416×416 for G++. But, the run-time with OpenMP is much slower than without OpenMP for G++ (compare columns Normal and OpenMP Normal) because the auto-vectorization is performed for Normal but not for OpenMP Normal.

Table 2: The run-times of matrix multiplication (row-major)

	Normal	AVX	OpenMP Normal	OpenMP AVX
VC++ (4)	0.029821	0.004304	0.023691	0.004372
VC++ (2)	0.022238	0.005275	0.023987	0.004672
G++ (4)	0.026333	0.029333	0.066667	0.020333
G++ (2)	0.030667	0.030667	0.077667	0.025667

Table 3: The run-times of matrix multiplication (column-major)

	Normal	AVX	OpenMP Normal	OpenMP AVX
VC++ (4)	0.023848	0.004387	0.018558	0.004722
VC++ (2)	0.027695	0.004567	0.019088	0.004411
G++ (4)	0.093000	0.028333	0.055667	0.017667
G++ (2)	0.094000	0.026333	0.074333	0.020333

Now, look at Normal column in Table 2 and Table 3 for G++. The run-time for column-major is about 3 times slower than for row-major. The reason is that the auto-vectorization is carried out for row-major, but not for column-major.

It can be seen that the run-time is greatly reduced when AVX intrinsic functions are used for both VC++ and G++. OpenMP is unable to improve performance a lot. This may be because the processor has only 2 cores.

7 Conclusion

This study is the continuation of our previous work [3] that illustrates how programming can be performed at different levels. This work illustrates how and in what situation programmers should use intrinsics and pragmas explicitly to perform a loop vectorization and parallelization, respectively, or otherwise, let compilers do the job automatically. Based on this work the following conclusions can be drawn:

- Vectorization using intrinsics is necessary for non-trivial cases because auto-vectorization is performed for simple cases only. G++, not VC++, can vectorize the inner-loop of matrix multiplication when matrices are stored in row-major. But, both cannot vectorize the outer loop of matrix multiplication when matrices are stored in column-major.
- Performance with auto-parallelization is not as good as that with parallelization using Open-PM. Using Open-MP directives to parallelize a loop is easy.
- The performance gain with vectorization is significant with using both VC++ and G++ compilers.
- The performance gain with parallelization is not significant with using both VC++ and G++ compilers. This may be because the cache misses and the memory bandwidth limitations.
- Selecting right algorithm and right data storage format is important because matrix multiplication with row-major storage is 3-times faster than column-major when using G++ compiler.

The authors will investigate more realistic and challenging algorithms for the performance gain by using vectorization and parallelization.

8 References

[1] Chris Lomont, "Introduction to Intel® Advanced Vector Extensions", available at https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions

[2] Microsoft, "Vectorizer and Parallelizer Messages", available at https://msdn.microsoft.com/en-us/library/jj658585(v=vs.120).aspx

[3] Xuejun Liang, Loretta A. Moore, and Jacqueline Jackson, "Programming at Different Levels: A Teaching Module for Undergraduate Computer Architecture Course", in Proceedings of the 2014 International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS'14), pp.77-83, Las Vegas, Nevada, USA, July 21-24, 2014