

# Comparison of compiler efficiency with SSE and AVX instructions

Maria Pashinska-Gadzheva  
Institute of mathematics and  
informatics  
Bulgarian academy of sciences  
Veliko Tarnovo, Bulgaria  
mariqpashinska@math.bas.bg, 0000-  
0003-3489-7744

**Abstract**—In this paper, we present a comparison between the execution times of code generated by three widely used compilers for C/C++ for specific class of problems connected to our research. We juxtapose the execution time of the software compiled by clang, gcc, and msvc on Windows OS and Ubuntu OS. The software has explicit use of the SSE4.1 and AVX2 instruction sets and calculates the weight spectrum of a linear  $[n,k]$  code over the finite field  $GF(2)$ . The build platform used for the projects is CMake which allows us to easily control the compilation parameters. The conclusions are made based on efficiency of the compiler

**Keywords**— C/C++ compilers, Optimization of mathematical software, Linear Codes, Weight Distribution

## I. INTRODUCTION

Many problems in coding theory and combinatorics need an extreme amount of computer computations. Therefore, the software implementing the algorithms needs to be able to execute all calculations in manageable time. There are many ways to optimize a software. The most popular ones are choosing better hardware, using a parallel implementation or the extended vector registers in modern central processing units (CPUs). However, a good optimization of an existing software can also be accomplished by the compiler itself. The compilers can implement many different types of optimization for both better execution times and space. Hence, the use of a compiler is of great importance for the efficiency of any software. There are different levels and approaches of the compiler for optimization of the execution time of a program as can be seen in [1]. It is possible for the compiler to integrate the upper mentioned extended registers into the code in some cases. However, a better approach is to explicitly tell it how to use the registers through the SSE and AVX instruction sets [2]. The goal of this paper is to compare the execution times of a software compiled with three different widely used C/C++ compilers - gcc [3], msvc [4], clang [5]. The software itself uses the extended registers and the SSE4.1 and AVX2 instruction sets and will soon be available online. A secondary goal of the paper is to present an easy way to manage the compilation process for the different compilers using the CMake program [6].

## II. PRELIMINARIES

The software we use to compare different compilers is made for our research purposes. It implements an algorithm for calculating the weight spectrum of a binary linear  $[n,k]$  code

over  $GF(2)$ . Linear codes are typically used to detect and/or correct errors occurring while transmitting information on a noisy channel. A binary linear code  $C$  is every  $k$ -dimensional subspace of the  $n$ -dimensional vector space over the field with two elements. The elements of the code are called *codewords*. The parameters  $n$  and  $k$  are length and dimension of the code, respectively. The codewords of  $C$  are calculated by listing all linear combinations of the rows of a matrix  $G$ , where the rows of  $G$  are  $k$  linearly independent vectors form  $C$ . This matrix is called *generator matrix* of the code. One of the most important parameters of a linear code is its *minimal distance*  $d$ , which is the minimum distance amongst the distances of all pairs of different codewords. The distance between two words is the number of positions in which they differ. There are different strategies for calculating  $d$ . One approach is to use the Brouwer–Zimmermann algorithm [7], [8], which is appropriate for cases where the code has large  $k$  and small  $n$ . The *weight spectrum* of the code is used in the general case when the dimension of the code is small and the length of the code is large. The weight spectrum of  $C$  is the sequence  $A_1, A_2, \dots, A_n$ , where  $A_i$  is the number of codewords with Hamming weight  $i$  (the Hamming weight of the vector is defined as the number of its nonzero elements). The weight spectrum is important since it not only can be used to calculate the minimum distance of a random linear code but also many other of its algebraic properties [9]. These properties give information about the efficiency of error correction and error detection of the code. There is an algorithm for calculation of the weight spectrum of a code without generating the codewords themselves using Walsh transform [10], [11]. This algorithm uses a characteristic vector of the code to calculate Walsh coefficients and these coefficients are used to calculate the weight spectrum. In the general case, however, we need all codewords to calculate it. There are different algorithms to calculate all codewords of a binary linear code. One approach, described in [12], uses an additional matrix, which stores a linear combination of exactly  $j$  rows of the generator matrix  $G$  in row  $j$ . It can be used for codes over  $GF(q)$ , where  $q$  is a prime power. For the binary case we use the transitional sequence of a binary Gray code [13], [14]. A binary Gray code of length  $m$  generates all  $2^m$  vectors of length  $m$  over the field with two elements by changing the previously generated vector in a single position. The transitional sequence of the Gray code gives the position where the change occurs. For generating the codewords of a

---

This work is partially supported by the Bulgarian National Science Fund under Contract No KP-06-Russia/33/17.12.2020.

linear code  $C$  the transition sequence shows the next row of  $G$  to add to the previous linear combination [15]. Table I gives an example for linear codes with  $k = 3$  and generator matrix  $G = (g_1, g_2, g_3)^T$  using the transition sequence  $T$  of a binary Gray code of length 3.

TABLE I.  
EXAMPLE OF GENERATION OF LINEAR COMBINATION USING  
GRAY CODE

| $T_i$ | GENERATION OF NEXT LINEAR<br>COMBINATION | RESULT            |
|-------|--|-------------------|
| 1     | $lc = 0 + g_1$                           | $g_1$             |
| 2     | $lc = (g_1) + g_2$                       | $g_1 + g_2$       |
| 1     | $lc = (g_1 + g_2) + g_1$                 | $g_2$             |
| 3     | $lc = (g_2) + g_3$                       | $g_2 + g_3$       |
| 1     | $lc = (g_2 + g_3) + g_1$                 | $g_1 + g_2 + g_3$ |
| 2     | $lc = (g_1 + g_2 + g_3) + g_2$           | $g_1 + g_3$       |
| 1     | $lc = (g_1 + g_3) + g_1$                 | $g_3$             |

### III. TESTING METHOD

The testing method has two main points to itself: the implementation of the algorithm and the control of the compilation process for the different compilers. For the implementation a bitwise representation of the codewords is used. Each codeword is written in  $x$  computer words, where  $x$  is the largest integer less than or equal to  $((n-1)/64 + 1)$ . The addition of two vectors is executed by the bitwise operation  $XOR$ . The main point of the C++ implementation is the model of work of the registers - the idea is to store the elements of the vector into 128 or 256-bit registers. The SSE and AVX instruction sets give a set of functions that are executed over the 128 and 256-bit registers, respectively. These functions have different execution time according to their implementation. Some functions are implemented by more than one CPU command, which can result in a slower execution time compared to others. The most used instructions in our case are the `_mm_xor_si128` and `_mm256_xor_si256` functions that translate into a single CPU command and execute XOR operation over all bits in the registers at the same time. The bitwise representation of the vector allows us to store 128 elements in a register with length 128 bits. Then the XOR operation will be executed over all elements at the same time. If the vector is of length 200 we can use two 128-bit registers or one 256-bit register. For codes with  $n \leq 64$  this representation of the codewords will result in insufficient use of memory. To avoid this we use  $[n, k-1]$  code  $C'$  with generator matrix  $G'$ , obtained from  $G$  by removing the  $k$ -th row  $g_k$ . We use a coset of  $C'$  determined by the vector  $g_k$  (coset of the code is the set  $g_k + C' = \{g_k + c | c \in C'\}$ ). Hence, the linear code  $C$  is equal to  $C' \cup (C' + g_k)$ . For the implementation we save the vector  $g_k$  in the second half of a 128-bit register. We also need to store the generator matrix  $G'$  as follows: each row will be duplicated into the two halves of a 128-bit register. Thus, we generate the codewords of the code  $C'$  and the vectors of the coset  $g_k + C'$  simultaneously. For example, the first linear combination can be calculated using the formula  $lc = lc \oplus r1$ . The initial vector will be  $lc = (0, 0, \dots, 0, g_k)$  stored in 128-bit register. The first row of the matrix is also stored in 128-bit register  $r1 = (g_1, g_1)$ . The next linear combinations are calculated by adding a row of the generator matrix  $G$  to the current vector  $lc$ .

To calculate the weight of a vector we use a function integrated in modern CPUs known as *popcnt*. This function returns the number of the non-zero bits of a 64-bit computer word. It has different syntax for the msvc and clang/gcc compilers and unlike other operations it does not have an implementation for larger registers. We use predefined macros in C/C++ to see which compiler is used. We also check if the function is available in the CPU by checking the 23-rd bit of special dedicated register using the `__cpuid` function for msvc and the `__get_cpuid` function for gcc/clang. Intel processors have four dedicated register for storing information about the CPU [16]. These registers also give the information about which extended instruction set is present in the CPU.

The second part of the testing method is management of the compilation process. The chosen compilers are designed and optimized for different operating systems and integrated development environments (IDEs). The term used for the compiler, running by default on an IDE is *native compiler*. For example, the msvc compiler is designed for Microsoft Visual Studio and it cannot be used with different IDEs. Furthermore, this IDE can only work with compatible compilers. The clang compiler, native for XCode IDE for Mac OSX [17], is such a compiler. It works through a special toolset developed for Visual Studio 2019. An important thing to know when using clang with Visual Studio is that the predefined macros for the compiler and the optimization flags are the same as the native compiler. In other words, clang is recognised by Visual Studio as native. The other compiler that we use is gcc. It is traditionally used on Linux/Unix based OS. However, it has an implementation for cross compilation on Windows OS known as mingw [18]. It is native for the CodeBlocks IDE [19] on both operating systems. All of the above shows that for the testing on Windows OS we need at least two different IDEs - Visual Studio and Codeblocks. Moreover, each compiler has different syntax for optimization flags. We use CMake to equalize those parameters.

CMake can be considered a build software for C/C++ even though its main functionality is to generate projects for different IDEs from the same source files. It is developed for most major operating systems (OSX, Ubuntu, Windows) and the accompanying IDEs (Xcode, Codeblocks, Visual Studio, etc.) and build systems (Make [20]). CMake gives the opportunity to choose one of the available IDEs for the target operating system. A main advantage is the possibility to change the compiler, whenever possible, set compiler flags, and manage others parameters through cache variables of the program. CMake has its own scripting language that is used for controlling those variables and describing the structure of the target software. It also can notify if a key component for the compilation is missing such as the compiler or an essential library. We use this software to generate projects for the three compilers. For Windows OS we generate two projects for Visual Studio using msvc and clang and a project for Codeblocks and mingw. The project using clang is generated by specifying the use of ClangCL toolset for Visual Studio before the generation of the project. For Ubuntu OS we simply generate Makefiles for the gcc and clang compilers. These files are used by GNU Make tool, which is a tool for controlling the generation of executables on Linux/Unix based operating systems. Our software uses SSE4.1 and

AVX2 instruction sets, hence the flags `-mavx2` and `/arch:AVX2` are added for mingw and msvc/clang compilers on Windows, respectively. On Ubuntu we only need `-mavx2` flag for both gcc and clang. These flags can also be added in the IDEs and Makefiles themselves. However, every IDE has different structuring of the properties of the compiler and it can be hard to add them correctly. Therefore, the use of CMake can be of great help. The projects are also only generated for building in release mode. This tells the compiler to optimize the code for faster execution time.

#### IV. EXPERIMENTAL RESULTS

In tables II and III we show the average execution times in seconds after 10 repeats for different  $n$  and  $k$  on Windows OS, while IV shows the average execution times in seconds after 10 repeats on Ubuntu OS. Table II gives average execution times for the implemented algorithm using 128-bit registers, while table III gives average execution times for 256-bit registers. We have executed the experiments 10 times in order to get more accurate times for the execution in general. These experiments are conducted on Intel Core i5 - 1035G1 CPU with 1.00 GHz clock frequency and on Windows 10 OS. We generate random codes for the given parameters for which we calculate the weight spectrum for all experiments. The four different lengths are chosen since for each of them a different number of registers of the given length is used. It can be seen that no compiler is significantly faster than the others. Also, we cannot definitively say which compiler is better for our specific problem on Windows OS - different compilers perform better for different parameters and registers. The results for the  $k = 30$  on Windows OS are also summarized in fig. 1 and fig. 2 since this case best describes the general conclusions for the all experiments. Some of the observations that can be made after analysing the experimental results are the following:

- Clang compiler used with Visual Studio gives worse execution times in most of the cases when compared to msvc.
- The msvc compiler performs better than the mingw compiler when using 128-bit registers in most cases. For the cases when it is slower, the difference in execution times is under 10%.
- The mingw compiler gives better execution times than both msvc and clang when using 256-bit registers and for

codes with smaller length. However, with the increase of the length of the code the execution time for msvc and clang improves compared to mingw.

- The execution time of the software compiled with msvc using 256-bit registers is slower compared to the 128-bit version for codes with  $n = 50$ . The difference in this particular case is that we write into memory the weight of four 64-bit computer words since we are calculating 4 codewords at the same time using cosets.

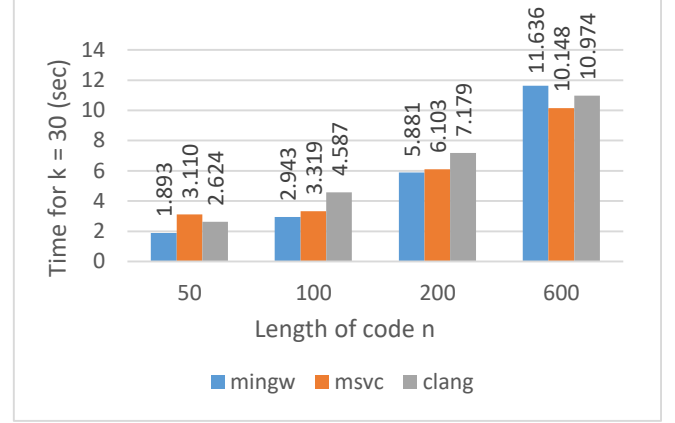


Figure 2: Average execution times using 256-bit register and Windows 10 OS for  $k = 30$

Table IV shows experimental results executed on Ubuntu 20.04 with Intel core i3-8145U with 2.1 GHz clock frequency. The parameters of the codes are shown in the first two columns and are the same as the experiments on the Windows OS. The next two columns show the average execution times in seconds after 10 repeats for calculation using 128-bit register and the compiled software with g++ and clang++ compiler. The last two columns show the average execution time in seconds for calculation using 256-bit register and the same compilers. The Make tool is used for compilation and the files are again generated using CMake. Thus, the same properties should be applied as the Windows versions. The conclusions that can be made are analogues to the ones for Windows. The differences in the average execution times is  $\pm 10\%$  between the two compilers. The clang++ compiler performs better when working with 128-bit registers, while g++ gives better results for the larger registers. Also, both compilers don't get as larger improvements of execution times for  $n = 600$  while using 256-bit registers compared to the 128-bit versions. The above observations, along with the implementation of the code, point to the conclusion that a software, compiled with msvc on Windows OS can result in slower execution times when frequent access to memory is needed. It is important to mention that there are compiler flags that favour optimizations for size over optimizations for speed for all compilers. Further research can be conducted into the efficiency of different C/C++ compilers since the work in this direction is limited. For different problems and operating systems the above conclusions may not be applicable as seen in [21], [22].

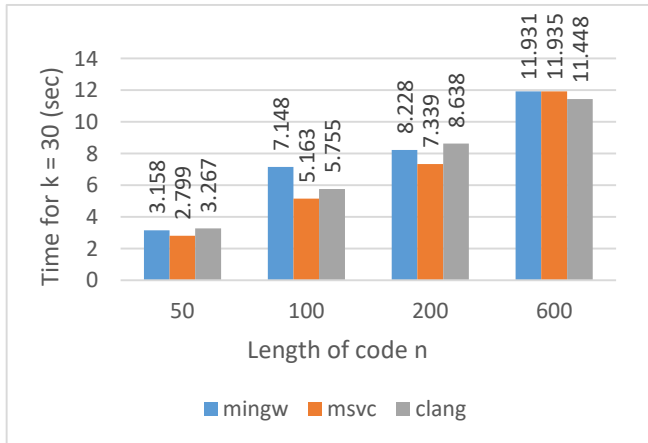


Figure 1: Average execution times using 128-bit register and Windows 10 OS for  $k = 30$

TABLE II.  
EXECUTION TIMES USING 128-BIT REGISTERS

| n   | k  | mingw  | msvc   | clang  |
|-----|----|--------|--------|--------|
| 50  | 26 | 0.202  | 0.186  | 0.234  |
|     | 28 | 0.795  | 0.693  | 0.799  |
|     | 30 | 3.158  | 2.799  | 3.267  |
| 100 | 26 | 0.323  | 0.335  | 0.396  |
|     | 28 | 1.126  | 1.303  | 1.451  |
|     | 30 | 7.148  | 5.163  | 5.755  |
| 200 | 26 | 0.452  | 0.483  | 0.551  |
|     | 28 | 1.786  | 1.917  | 2.160  |
|     | 30 | 8.228  | 7.339  | 8.638  |
| 600 | 26 | 0.753  | 0.759  | 0.732  |
|     | 28 | 3.199  | 2.972  | 2.838  |
|     | 30 | 11.931 | 11.939 | 11.440 |

TABLE III.  
EXECUTION TIMES USING 256-BIT REGISTERS

| n   | k  | mingw  | msvc   | clang  |
|-----|----|--------|--------|--------|
| 50  | 26 | 0.134  | 0.182  | 0.157  |
|     | 28 | 0.483  | 0.725  | 0.603  |
|     | 30 | 1.893  | 3.11   | 2.624  |
| 100 | 26 | 0.195  | 0.215  | 0.280  |
|     | 28 | 0.741  | 0.831  | 1.139  |
|     | 30 | 2.943  | 3.319  | 4.587  |
| 200 | 26 | 0.372  | 0.396  | 0.438  |
|     | 28 | 1.495  | 1.534  | 1.751  |
|     | 30 | 5.881  | 6.103  | 7.179  |
| 600 | 26 | 0.736  | 0.648  | 0.697  |
|     | 28 | 2.919  | 2.757  | 2.861  |
|     | 30 | 11.636 | 10.148 | 10.974 |

## V. CONCLUSION

The three compared compilers are the most popular ones for C/C++ programs. Nowadays, they are designed to work with specific IDEs and operating systems and switching from one to another may be a difficult task. Furthermore, different results may be expected based on the algorithm, its implementation, the platform, the operating system and many other factors. A conclusion that can be made for our specific software working with SSE4.1 and AVX2 instruction sets is that using Visual Studio with msvc compiler is a good option for achieving good execution speed. The IDE can also easily be used with CMake to integrate the clang compiler for cross compilations, if needed, and is frequently preferred for its ease in using the debugger. The CMake software is also a very good tool to control most environment variables such as optimization flags and to transfer software between operating systems. For Ubuntu OS and our specific software the gcc compiler is more appropriate since it is typically used with most IDEs. For the smaller registers the differences in execution time compared to the clang version is small enough that it can be neglected.

## REFERENCES

- [1] A. Fog, "Optimizing software in C++ An optimization guide for Windows, Linux and Mac platforms", online, 2004, available at: [http://www.agner.org/optimize/optimizing\\_cpp.pdf](http://www.agner.org/optimize/optimizing_cpp.pdf)
- [2] Intel® Intrinsics Guide, online at: <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>
- [3] GCC online documentation, online at: <https://gcc.gnu.org/onlinedocs/>
- [4] Microsoft documentation for C/C++ projects and build systems in Visual Studio, online at: <https://docs.microsoft.com/en-us/cpp/build/projectsand-build-systems-cpp?view=msvc-170>

TABLE IV.  
EXECUTION TIMES FOR 128 AND 256 BIT REGISTER ON UBUNTU OS

|     |    | 128-bit register |         | 256-bit register |         |
|-----|----|------------------|---------|------------------|---------|
| n   | k  | g++              | clang++ | g++              | clang++ |
| 50  | 26 | 0.204            | 0.195   | 0.116            | 0.123   |
|     | 28 | 0.849            | 0.766   | 0.470            | 0.517   |
|     | 30 | 3.156            | 2.964   | 1.780            | 2.504   |
| 100 | 26 | 0.311            | 0.349   | 0.176            | 0.213   |
|     | 28 | 1.068            | 1.280   | 0.687            | 0.819   |
|     | 30 | 4.304            | 5.490   | 2.745            | 3.265   |
| 200 | 26 | 0.401            | 0.291   | 0.359            | 0.369   |
|     | 28 | 1.572            | 1.523   | 1.488            | 1.444   |
|     | 30 | 6.279            | 6.047   | 5.687            | 6.203   |
| 600 | 26 | 0.710            | 0.626   | 0.655            | 0.707   |
|     | 28 | 2.803            | 2.427   | 2.592            | 2.817   |
|     | 30 | 11.188           | 9.748   | 10.323           | 11.474  |

- [5] Clang: a C language family frontend for LLVM, online at: <https://clang.llvm.org/>
- [6] CMake Reference Documentation, online at: <https://cmake.org/cmake/help/latest/>
- [7] A. Betten, M. Braun, H. Friepertinger, A. Kerber, A. Kohnert, A. Wassermann, "Error-Correcting Linear Codes: Classification by Isometry and Applications", Springer: Berlin/Heidelberg, Germany, 2006.
- [8] S. Bouyuklieva, I. Bouyukliev, "An Extension of the Brouwer–Zimmermann Algorithm for Calculating the Minimum Weight of a Linear Code", Mathematics, vol. 9, no. 19, pp. 2354, 2021.
- [9] R. Dodunekova, S. M. Dodunekov, "Sufficient conditions for good and proper error-detecting codes", IEEE Transactions on Information Theory, vol. 43, no. 6, pp. 2023–2026, 1997.
- [10] I. Bouyukliev, St. Bouyuklieva, T. Maruta, P. Piperkov, "Characteristic vector and weight distribution of a linear code", Cryptography and Communications, vol. 13, pp. 263–282, 2021.
- [11] M. Pashinska, I. Bouyukliev, "A parallel algorithm for computing the weight spectrum of binary linear codes", 2020 Algebraic and Combinatorial Coding Theory (ACCT), pp. 118–122, 2020.
- [12] I. Bouyukliev and V. Bakoev, "A method for efficiently computing the number of codewords of fixed weights in linear codes" Discrete Applied Mathematics, vol. 156, pp. 2986–3004, 2008.
- [13] F. Gary, "Pulse Code Communication", New York, USA: Bell Telephone Laboratories, Incorporated. U.S. Patent 2,632,058. Serial No. 785697.
- [14] D. Knuth, "The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1", Addison-Wesley Professional, 2014.
- [15] T. Gulliver, V. K. Bhargava, and J. M. Stein, "Q-ary Gray codes and weight distributions," Applied Mathematics and Computation, vol. 103, pp. 97–109, 1999.
- [16] Intel® 64 and IA-32 Architectures Software Developer Manuals online at: <https://software.intel.com/content/www/us/en/develop/articles/intelsdm.html>
- [17] Xcode Development Documentation, online at: <https://developer.apple.com/documentation/xcode>
- [18] MinGW Documentation Resources, online at: <https://www.mingw64.org/>
- [19] Codeblocks User Manual, online at: <https://www.codeblocks.org/usermanual/>
- [20] GNU Make documentation, online at: <https://www.gnu.org/software/make/>
- [21] M. Pashinska, I. Bouyukliev, "About OpenMP and some combinatorial algorithms", Mathematics and Education in Mathematics, vol. 49, pp. 167–172, 2020.
- [22] T. Botor, H. Habiballa, "Compiler optimization for scientific computation in C/C++, AIP Conference Proceedings, vol. 2040, pp. 030004, 2018.