Multidiszciplináris tudományok, 12. kötet, 1. sz. (2022), pp. 79-89 **https://doi.org/10.35925/j.multi.2022.1.7**

IMPROVING PERFORMANCE WITH SIMD INTRINSICS

Péter Mileff
associate professor, University of Miskolc, Department of Information Technology, 3515 Miskolc, Miskolc-Egyetemváros, e-mail: mileff@iit.uni-miskolc.hu

Abstract
Computer application development has a long history. With the continuous development of hardware, both the software and the programming languages and frameworks that make application development possible have also changed. As we move forward in time, as more and more powerful computers become available, the level of programming languages has also risen to a higher level. Increasingly high-level APIs and languages have become available, pointers, memory allocations and releases have slowly disappeared from the area of the programming. However, with the transition to high-level languages, the optimization of programs or the possibility of it has been relegated to the background, because some efficiency-enhancing options are not available in high-level languages. This paper therefore delves deeper into the benefits of SSE/AVX based SIMD concurrency capabilities that are "forgotten" in everyday programming. The presented tests will show that there are reserves in the CPU that can be used to push certain performance limits further away.

Keywords: *SIMD, optimization, SSE, AVX*

1. Introduction

Maximizing the performance of a given hardware architecture by precisely optimizing running software has always been a key point in the history of computer science. Of course, in the beginning, when the first personal computers appeared, these issues were far more pressing than they are today as those computers had rather poor computing capacity. However, the computer gaming industry has seen these as opportunities from the beginning and the gaming world, which is now gigantic, has started. Since the hardware wasn't powerful, and the GPU didn't even exist yet, it took a lot of attention to develop a game or other software that was already intended to perform more powerful calculations. Especially in 3D games where everything happens in real time.

The C language was already available, which was still considered high-level at the time. Most of the programs are made for this because of speed criticality. However, that alone was not enough. Due to limited hardware, programmers used languages that were close to machine languages to create critical parts. Computer games were a typical example of this line. The heart of the rendering engine, vectorial and other critical calculations were usually made in Assembly and later with a SIMD (Single Instruction Multiple Data) instruction set. Several languages have provided very good opportunities for this. We were able to insert even 4-5 lines of Assembly code inside a for loop to reach the proper speed. Quite great results have been achieved due to its precise and in-depth optimization, due to the appropriate level of hardware utilization.

With the development of technology, the hardware has become more powerful. Higher level languages have appeared and spread: Java, C #, Javascript, Scala, SWIFT, etc. Today, it is only natural to build applications with these technologies. So the development of hardware was followed by the development of software. Higher-level languages have increased industrial productivity. Programs could be created faster in a single unit of time, as the programmer was no longer necessarily focused on pointers or releasing memory, but solely on business logic. The industry is, of course, very happy with these technologies, as software is often made faster this way. However, the big disadvantage of this line of development is that the software is often not properly optimized. It requires much more powerful hardware than it would actually require. All of this is because software made with higher-level languages has more and more dependencies, more and more layers on top of each other, and many times the possibility of lower level programmability is missing. Because of the performance of today's hardware, most of the time, programmers no longer have the “compulsion” to optimize, and because we can no longer reach the level of proper optimization in the higher level languages.

The so-called SIMD instruction sets were already available in the early CPU (MMX, SSE), and in recent years the technology has further developed (NEON, AVX, AVX512). Initially, it was a holy grail for game developers, as a successful SSE-based optimization resulted in a large increase in performance. These made it possible for games like Doom, Quake, Unreal to run on the hardware at the time. If we are talking about gaming, we are now relying more on the GPU, which is not due to the fact that SIMD solutions are not suitable, but rather to the appearance of higher-level languages. For other programs that may require critical computation, multithreading is the primary solution.

The direct use of SIMD solutions has been supplanted by the day-to-day work of the average programmer. This is because SIMD solutions require a higher level of knowledge and trust in the compiler. Today's compilers are becoming more advanced, able to compile code using the SIMD extended instruction set. While this is a great thing, in most cases it does not compete with direct SIMD codes.

In this paper, we investigate the SIMD-based parallelism solution, especially the SSE and AVX extensions. We created a self-made benchmark package to illustrate the potential speed increase provided by SIMD.

2. Related works

The SIMD based instruction set extension of the processor has a long history. Development is still ongoing today, though not as spectacular and media-centric as it is in the case of GPUs. The following are the most important milestones.

MMX™ technology MMX™ technology - Intel® Pentium Pentium® with MMX™ and Pentium with MMX™ and Pentium® II processors (1996): Introduced 64-bit MMX (Multimedia Extensions) registers for SIMD integer operations. Supports SIMD operations on packed byte, word, and double-word integers. Useful for multimedia and communications software.

3D NOW – AMD (1998): It adds SIMD instructions to the base x86 instruction set, enabling it to perform vector processing of floating-point vector-operations using Vector registers, which improves the performance of many graphic-intensive applications. The first microprocessor to implement 3DNow was the AMD K6-2, which was introduced in 1998. When the application was appropriate, this raised the speed by about 2-4 times.

SSE – Intel® Pentium Pentium® III processor (1999): Originally called Katmai New Instructions (KNI), because Katmai was the codename for the first Pentium III core revision. Introduced 128-bit extended memory manager (XMM) registers for SIMD integers and FP-SP operands. Executes FP and

SIMD simultaneously. Introduced data prefetch instructions. Useful for 3D geometry, 3D rendering, and video encoding/decoding.

SSE2 – Intel® Pentium Pentium® 4 and Intel 4 and Intel® Xeon processors ™ (2000): Added extra 64-bit SIMD integer support. Has the same XMM registers for SIMD integer and floating point double precision (FP-DP). Has 144 new instructions for data support (no new registers). Adds support for cacheability and memory ordering operations. Release was focused on 3D games, Computer-Aided Design applications and video encoding/decoding. Although SSE2 can operate over four elements, the performance was roughly the same as MMX, which operates in just two elements. This loss of performance is due to accessing misaligned memory addresses.

SSE3 – Intel® Pentium® 4 Processor (2004): In order to solve performance issues due to misaligned data, SSE3 incorporates new instructions to load from unaligned memory addresses minimizing timing penalties. Accelerates performance of Streaming SIMD Extensions technology, Streaming SIMD Extensions 2 technology and X87 technology, and X87-FP math capabilities. Useful in some 3D operations (Quaternions) complex arithmetic and video codec algorithms.

SSSE3 – Intel ® Core® 2 Processor (2006): Application performance improvement (new instructions such as multiply-and-add and vector alignment). Potential for specific application domains.

SSE4 (2007): SSE4 contemplates 54 new instructions (47 in SSE4.1 and 7 in SSE4.2) dedicated to string processing. Also includes elaborated instructions to perform population count and computation of CRC-32 error detecting code4. The final version of the SSE4 is SSE4.2.

The AVX extension

Intel decided to move computations to wider registers and introduced the Advanced Vector Extensions (AVX, also known as Sandy Bridge New Extensions). AVX is an extension to the x86 instruction set architecture for microprocessors from Intel and AMD proposed by Intel in March 2008 and first supported by Intel with the Sandy Bridge processor shipping in Q1 2011 and later on by AMD with the Bulldozer processor shipping in Q3 2011. AVX provides new features, new instructions and a new coding scheme.This technology involves:

- 16 256-bit registers, called YMM0-YMM15
- The ability to write three operand code in assembler listings
- Proposes the VEX encoding scheme that increases the space of operation codes
- It also supports the legacy SSEx instructions by adding the VEX prefix

The second version of AVX, released in 2012, includes the expansion of many integer operations to 256-bit registers. AVX2 supports gather/scatter operations to load/store registers from/to non-contiguous memory locations. A big confusion was caused about future directions of new instruction sets, both Intel and AMD changed their plans about SSE5. Bulldozer, AMD’s latest micro-architecture, implements XOP and FMA4 instruction set, and also has compatibility with AVX. Piledriver and Haswell are the next micro-architectures from AMD and Intel, respectively. They will provide more multiply-and-add instructions for both floating point and integer operations.

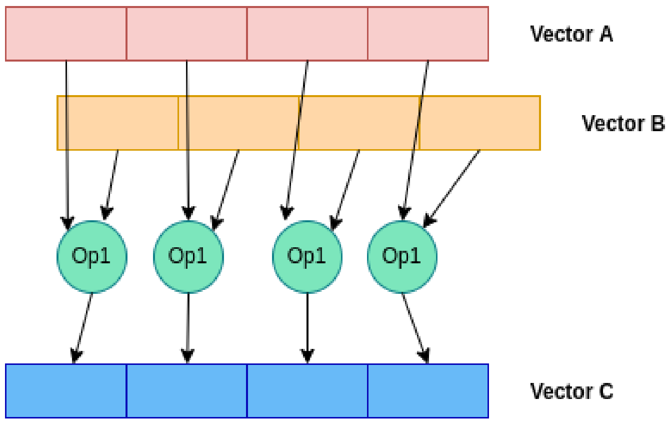
In 2011, ARM has also introduced SIMD extensions to ARM-Cortex architectures with their NEON technology. The NEON SIMD unit is 128-bit wide and includes 16 128-bit registers that can be used as

32 64-bit registers. These registers can be thought as vectors of elements of the same data type, being the data types signed/unsigned 8, 16, 32, 64-bit, and single precision floating point.

3. SIMD overview

SIMD units have been available in Intel microprocessors since the advent of the MMX, SSE, and AVX ISA extensions. The MMX extensions were initially included to speed up the performance of multimedia applications and other application domains requiring image and signal processing. Single instruction, multiple data (SIMD), is a class of parallel computers in Flynn's taxonomy (David P, 2011). It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. In contrast, the conventional sequential approach using one instruction to process each individual data is called scalar operations. In short, SIMD allows for processing several data values with one single instruction. Thus, such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment. SIMD is particularly applicable to common tasks like adjusting the contrast in a digital image or adjusting the volume of digital audio. In order to improve the performance of multimedia use most modern CPU designs include SIMD instructions.

Typically, a SIMD unit receives as input two vectors (each one with a set of operands), performs the same operation on both sets of operands (one operand from each vector), and outputs a vector with the results (João et al, 2017). *Figure 1* illustrates a simple example of a SIMD unit executing four operations in parallel:

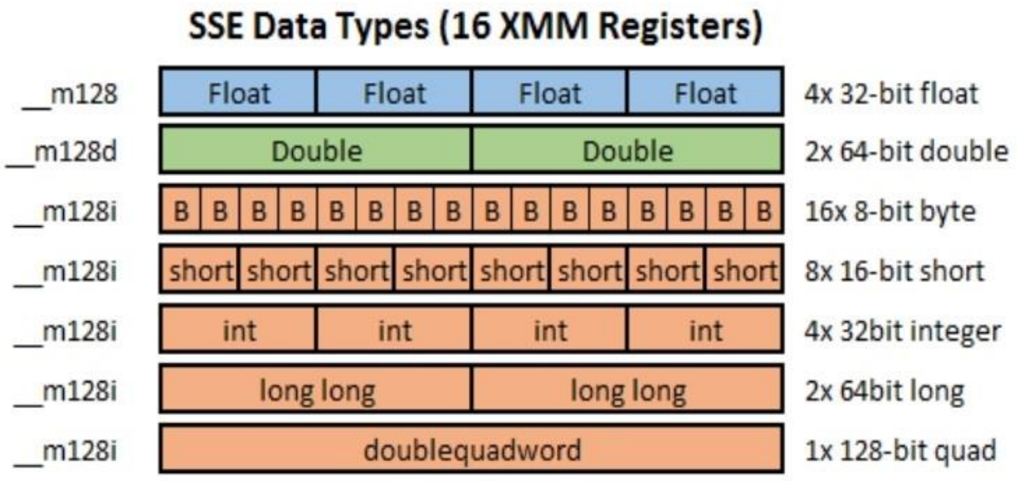


1. Figure. *SIMD operation execution*

3.1 SSE registers

There are 8 (16 in 64-bit mode) XMM registers [XMM0-7(15)] that come with SSE, and they are 128-bit registers. A typical 128 bit SIMD register can contain:

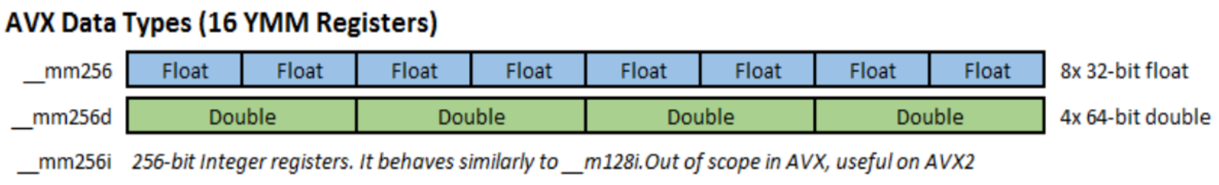
- sixteen 8 bit integer values (int8x16 and uint8x16)
- eight 16 bit integer values (int16x8 and uint16x8)
- four 32 bit integer values (int32x4 and uint32x4)
- four single precision floating point values (float32x4)
- two double precision floating point values (float64x2)



2. Figure. *SSE registers* [4]

3.2 The AVX extension

AVX uses sixteen YMM registers to perform a Single Instruction on Multiple pieces of Data (see SIMD). Each YMM register can hold and do simultaneous (math) operations on: eight 32-bit single-precision floating point numbers or four 64-bit double-precision floating point numbers. The width of the SIMD registers is increased from 128 bits to 256 bits, and renamed from XMM0–XMM7 to YMM0–YMM7 (in x86-64 mode, from XMM0–XMM15 to YMM0–YMM15).


AVX Data Types (16 YMM Registers)
__mm256
Float
8x 32-bit float
__mm256d
Double
4x 64-bit double
__mm256i
256-bit Integer registers. It behaves similarly to __m128i.Out of scope in AVX, useful on AVX2

3. Figure. *AVX registers* [4]

Via the VEX prefix the legacy SSE instructions can be still utilized to operate on the lower 128 bits of the YMM registers. AVX introduces a three-operand SIMD instruction format called VEX coding scheme, where the destination register is distinct from the two source operands. For example, an SSE instruction using the conventional two-operand form $x = x + y$ can now use a non-destructive three-operand form $z = x + y$, preserving both source operands. Originally, AVX's three-operand format was limited to the instructions with SIMD operands (YMM), and did not include instructions with general purpose registers (e.g. EAX). It was later used for coding new instructions on general purpose registers in later extensions, such as BMI. VEX coding is also used for instructions operating on the k0-k7 mask registers that were introduced with AVX-512 (Intel, 2011).

The new VEX coding scheme introduces a new set of code prefixes that extends the opcode space, allows instructions to have more than two operands, and allows SIMD vector registers to be longer than

128 bits. The VEX prefix can also be used on the legacy SSE instructions giving them a three-operand form, and making them interact more efficiently with AVX instructions without the need for VZEROUPPER and VZEROALL (Intel, 2011). The AVX instructions support both 128-bit and 256-bit SIMD. The 128-bit versions can be useful to improve old code without needing to widen the vectorization, and avoid the penalty of going from SSE to AVX, they are also faster on some early AMD implementations of AVX. This mode is sometimes known as AVX-128.

4. Use SIMD in practice

SSE/AVX enabled CPU's have assembler instructions for operating with XMM and YMM registers. But in most compilers the process is simplified by using *intrinsic* functions, so programmers don't need to use assembly directly. Compilers wrap assembler instructions as functions in order to use them as easily as calling a function with the right parameters. Sometimes these intrinsic functions are emulated if the CPU doesn't support the instruction set.

Using the intrinsics is no different than using any other C/C++ library. The programmer includes the correct header file for the type of intrinsic to be used, and then calls the desired intrinsic function. Most compilers have some level of auto vectorization / SIMD support.

In order to use SSE/AVX, we need to have the proper architecture. The SSE/AVX compiled binaries will not run on machines without AVX capabilities. In order to support different CPUs, platform specific binaries should be compiled separately.

SSE/AVX intrinsic functions use the following naming convention (Intel, 2011):

```
_<vector_size>_<intrin_op>_<suffix>
```

- <vector_size> is mm for 128 bit vectors (SSE), mm256 for 256 bit vectors (AVX and AVX2), and mm512 for AVX512.
- <intrin_op> Declares the operation of the intrinsic function. E.g. add, sub, mul, etc.
- <suffix> Indicates the datatype. *ps* is for float, *pd* for double, and *ep<int_type>* is for integer datatypes: *epi32* for signed 32 bit integer, *epu16* for unsigned 16 bit integer, etc.

SSE adds three typedefs: *__m128* , *__m128d* and *__m128i*. Float, double (d) and integer (i).
AVX typedefs: *__m256* , *__m256d* and *__m256i*. Float, double (d) and integer (i).
Important notice is that XMM and YMM overlap. XMM registers are treated as the lower half of the corresponding YMM register. This can introduce some performance issues when mixing SSE and AVX code. In the level of the programming *__m128i* and *__m256i* are unions, so the datatype needs to be referenced. GCC allows for access to data components as an array.

Using intrinsic functions is compiler dependent. In order to find all the intrinsic functions, the compiler documentation should be read.

4.1 Detect System Capabilities

In this paper, we focus on the *GCC (GNU Compiler Collection)* compiler, which is widely used in practice. Any further information is related to this compiler, however they are almost true for the most known compilers as well.

```
#!/bin/bash
# CPU flag detection
echo "****Getting CPU flag capabilities and number of cores"
cat /proc/cpuinfo | egrep "(flags|model name|vendor)" | sort | uniq -c
# Compiler capabilities
echo "****Getting GCC capabilities"
gcc -march=native -dM -E - < /dev/null | egrep "SSE|AVX" | sort
```

In the CPU flag capabilities, we should search for the SSE or/and AVX flag which identifies the CPU compatibility. If we have AVX2 that means the CPU allows AVX2 extensions. In the GCC capabilities we'll search for the **#define __AVX__1** pragma, which indicates that the AVX branches will be enabled.

If we run GCC without the correct *march* flag we won't get the **__AVX__** flag, so therefore using the **-march=native** or **-mavx** flags are necessary. Default GCC parameters are generic, without the flag it won't enable AVX even if the CPU is AVX capable.

4.2 Compiler's Automatic Vectorization (CAV)

GCC is an advanced compiler, and with the optimization flags **-O3** or **-ftree-vectorize** the compiler will search for loop vectorizations (**-mavx** flag is needed also). It does not affect the source code, it remains the same, but the compiled code by GCC is completely different.

GCC won't log anything about automatic vectorization unless some flags are enabled. In order to get detailed informations, more compiler flags should can be used:

- **-fopt-info-vec or -fopt-info-vec-optimized**: The compiler will log which loops (by line N°) are being vector optimized.
- **-fopt-info-vec-missed**: Detailed info about loops not being vectorized, and a lot of other detailed information.
- **-fopt-info-vec-note**: Detailed info about all loops and optimizations being done.
- **-fopt-info-vec-all**: All previous options together.

4.3 Criteria for loop vectorization

Not all loops can be vectorized. In order for vectorization to be performed, there are some strict requirements for the loop.

- Constant loop count: the end of the loop can be a dynamic variable, increasing or decreasing its value at will, but once the loop starts, it must be constant.
- Limited loop control: usage of break or continue sentences are limited. Sometimes the compiler is clever enough to make it work, but in some cases the loop won't be vectorized.
- There are some limits on calling external functions inside a loop.
- Limited loop counter dependencies: there shouldn't be data dependencies with other indexes of the loop. For example: *for (int i=1; i < N; ++i) x[i]=x[i-1]*8;* is traversed with a variable *i*, and data *x[i]* depends on the previous *x[i-1]* value. Since AVX registers are loaded as 8 floats, the compiler can't do these calculations with a vector.
- Conditional sentence limitation: if/else can be used if they don't change the control flow, and are only used to conditionally load *A* or *B* values into a *C* variable.

The advantage of autovectorization is that the developer does not need to change anything, and maybe the loop will be vectorized. But sometimes (especially in high performance computing applications) loops and vectorization need to be fine tuned, ensuring maximum throughput by using manual SSE / AVX vectorization.

4.4 Missing functions in SSE/AVX intrinsics

4.4.1 Lack of integer division

SSE and AVX lack (real) integer division operators. In the Intel documentation (Intel, 2011), there are functions to perform integer division (e.g. _mm_idiv_epi32), but these intrinsics generate a sequence of instructions, which may perform worse than a native instruction.

There are some ways to overcome this:

- By calculating the division in linear code. Retrieving the single data, divide them and store again in the vector. This can be slow.
- Converting the integer vector to float, perform the division and convert back to integer.
- For known divisors at compile time, there are some magic numbers to convert division by a constant into a multiplication operation.
- For the power of two divisions, bit shifting operations can be used. This can only be done if all the vectors are divided by the same power of two numbers.

4.4.2 Lack of trigonometric functions

There aren't trigonometric functions in vector intrinsic functions. Possible solutions are calculating them with linear code (one by one for each vector value), or creating approximation functions.

4.4.3 Lack of a random number generator

Additionally, there aren't random number generators for vectors as intrinsics. But it's simple to recreate a good pseudorandom generator from a linear version. Just be sure about the bits used in the pseudo-random number generator. 32 or 64bit RNG are preferred for filling vectors.

4.5 Performance penalties

4.5.1 Data Alignment

Older CPU architectures can't use vectorization unless data is memory aligned to the vector size. Some other CPU's can use unaligned data with some performance penalties. In recent processors the penalty seems to be negligible, but just to be safe it could be a good idea to align data if it doesn't add excessive overhead.

In GCC, data alignment can be done with these variable attributes: ***__attribute__((aligned(16)))*** ***__attribute__((aligned(32)))***

4.5.2 Transition Penalties between SSE and AVX

There is another big problem when mixing legacy SSE libraries and the new AVX architecture. Because XMM and YMM share the lower 128 bits, transitioning between AVX and SSE can lead to undefined values in the upper 128bits. To solve this, the compiler needs to save the upper 128bits, clear it, execute the old SSE operation, and then restore the old value. This adds a noticeable overhead to AVX operations, resulting in reduced performance. This issue does not mean we can't use *__m128* and *__m256* at the same time without performance penalties. AVX has a new instruction set for *__m128*, with VEX prefixes. These new VEX instructions don't have any problem combining with *__m256* instructions. The transition penalty is when non-VEX *__m128* instructions are combined with *__m256* instructions. This happens when you use old SSE libraries linked into new AVX enabled programs.

To avoid transition penalties, the compiler can automatically add calls to *VZEROUPPER* (clears out the upper 128bits) or *VZEROALL* (clears out all the YMM register) with the *-mvzeroupper* parameter, or the programmer can do it manually. If we are not using external SSE libraries, and we are sure all our code is VEX-enabled and compiled with AVX extensions enabled, we can instruct the compiler to avoid adding *VZEROUPPER* calls, with: *-mno-vzeroupper*

4.5.3 Data loading, unloading and shuffling

Moving data back and forth from AVX registers can be expensive. In some cases, if we have some data stored in linear structures, sending this data to AVX vectors, executing some operations, and recovering this data is more expensive than simply performing calculations in a linear way. So programmers must take into account the data loading and unloading overhead indeed in some cases it becomes a bottleneck.

5. Test Results

A SIMD-based implementation can greatly increase the computational efficiency of a program. Of course, it is not possible to make universal statements, as it is highly dependent on the nature of the particular program (e.g. CPU intensive or not) as well as the SIMD implementation. Not all calculations can be successfully transformed into a SIMD implementation that will be quite efficient in performance. For this reason, to test the effectiveness of SIMD solutions, we chose basic math problems that are likely to appear in most CPU-intensive applications. The efficacy was measured in five test cases, where the proper SIMD version of the problem was implemented using SSE / AVX:

- Vector length calculation
- Dot product
- Vector-scalar multiplication
- Vector-vector multiplication
- Cross product

The programs were written in the C++ language using the GCC 11.1.0 compiler. The measurements were performed on an Intel (R) Core (TM) i7-9700 CPU @ 3.00GHz CPU using a single thread. Each mathematical case was tested with 400,000 4-element vectors [e.g. Vector4(4,3,5,1)]. The final results in the table were averaged over 5 runs. The values are measured in nanoseconds using the high resolution clock of the Linux operating system.

1. table. *Benchmark results*

	Naive	SSE	AVX	Performance increase SSE	Performance increase AVX
Dot product	3467776	2033760	1423108	41.35%	30%
Vector length	2500097	2128189	1742673	14.87%	18.11%
Vector multiplication by scalar	4777252	3873024	267869	18.92%	6.92%
Vector multiplication by vector	4539375	2920104	1359712	35.67%	53.43%
Cross product	5301049	3697134	3053370	30.25%	42%

The results show clearly the benefits of SIMD versions, the performance increases in case of SSE and AVX are impressive. The results were in line with expectations. But, it should be noticed that the SIMD conversion, especially AVX, was not trivial.

Typically, proper data preparation is required for a successful SIMD conversion. Especially in the case of AVX, as we can already work with quite large registers there. If they are not filled properly, the increase in speed may not be possible. While in the case of SSE all elements of the four-element vector "naturally" fit into the *__m128* register, in the case of AVX this can no longer be said of the *__m256* register. Thus, the AVX transformation required not only a formal code change but also a restructure of the loops.

Finally, we should not forget that the amount of materials available from the topic is significantly less for those who wish to deal with this optimization level.

6. Conclusion

In real life, only a small percentage of programmers encounter CPU extensions. Developing an average application often does not require this level of optimization. In addition, application development in higher-level languages in line with today's trends does not allow direct usage of the extended instruction set of modern CPUs. Of course, programming language compilers try to do this automatically, making usually a great job, but without human intelligence, in most cases this cannot be done completely. Performance lags behind the maximum possible.

SSE / AVX / Neon-based extensions can greatly improve computing performance. Just think of the early computer games that were able to achieve amazing performance (MMX and SSE only) even on slow CPUs at the time. However, this comes at a price. Solving a programming task will be more difficult, and SSE / AVX / Neon code modifications will require higher programming knowledge and a lower level programming language, which can reduce a company's productivity. And from a business standpoint, this does not fit into today's bustling and profit-oriented world. An acceptable compromise for companies is to recommend more powerful hardware for a given program than to optimize their software properly.

References

[1] João M. P. Cardoso, José Gabriel F. Coutinho, Pedro C. Diniz (2017). *Embedded Computing for High Performance*. Efficient Mapping of Computations Using Customization, Code Transformations and Compilation, pp. 17–56.

[2] Intel® 64 and IA-32 Architectures Software Developer Manuals. Published on October 12, 2016, updated May 18, 2018. Available: https://software.intel.com/en-us/articles/intel-sdm [Megtekintés: 06-január-2022].

[3] David Padua (2011). *Encyclopedia of Parallel Computing*. Springer-Verlag New York Inc.

[4] SSE & AVX Vectorization (2022). https://www.codingame.com/playgrounds/283/sse-avx-vectorization/sse-and-avx-usage

[5] Dan C. Marinescu (2018). *Cloud Computing: Theory and Practice*. Second edition, Morgan Kaufmann.

[6] Intel Corporation (2011). Intel® Advanced Vector Extensions Programming Reference. http://www.intel.com

[7] Intel Corporation (2014). Optimizing Performance with Intel® Advanced Vector Extensions, WHITE PAPER, Intel® Advanced Vector Extensions Processor Performance.

[8] Hwancheol Jeong, Sunghoon Kim, Weonjong Lee, Seok-Ho Myung (2012). Performance of SSE and AVX Instruction Sets, *30th International Symposium on Lattice Field Theory* (Lattice 2012).

[9] Agner, Software optimization resources. C++ and assembly. Windows, Linux, BSD, Mac OS X. URL http://www.agner.org/optimize/

[10] Hossein Amiri, Asadollah Shahbahrami (2020). SIMD programming using Intel vector extensions. *Journal of Parallel and Distributed Computing*, Volume 135, pp. 83–100.

[11] Hossein Amiri, Asadollah Shahbahrami, Angela Pohl, Ben Juurlink (2018). Performance evaluation of implicit and explicit SIMDization. *Microprocessors and Microsystems*, Volume 63, pp 158–168.

[12] AMD (2000), 3DNow! Technology Manual, Tech. rep.

[13] Peleg A., Weiser U. (1996). MMX technology extension to the intel architecture. *IEEE Micro*, 16 (4), pp. 42–50.

[14] Péter Mileff, Judit Dudra (2014). Advanced 2D Rasterization on Modern CPUs, Applied Information Science, Engineering and Technology: Selected Topics from the Field of Production Information Engineering and IT for Manufacturing: Theory and Practice. Series: *Topics in Intelligent Engineering and Informatics*, Vol. 7, Chapter 5, Springer International publishing, pp. 63–79.