Stable vector operation implementations, using Intels SIMD architecture

József Smidla, István Maros

University of Pannonia, 10. Egyetem Str., Veszprém, Hungary H-8200, smidla@dcs.uni-pannon.hu, maros@dcs.uni-pannon.hu

Most floating point computations, in particular the additive operations, can have an annoying side-effect: The results can be inaccurate (the relative error can be any large) which may lead to qualitatively wrong answers to a computational problem. For the most prevalent applications there are advanced methods that avoid unreliable computational results in most of the cases. However, these methods usually decrease the performance of computations. In this paper, the most frequently occurring vector-vector addition and dot-product operations are investigated. We propose a faster stable add method which has been made possible by the appropriate utilization of the advanced SIMD architecture. In this work we focus only on the widely used Intel systems that are available for most of users. Based on it, stable vector addition and dot-product implementations are also introduced. Finally, we show the performance of these techniques.

Keywords: Scientific computations, Numerical accuracy, Accelerating stable methods, Heuristics, Intel's SIMD architecture, Cache techniques

1 Introduction

Several pieces of software use vector operations like vector addition and dot product. Scientific applications require double precision floating point number representation because of the required accuracy of the computations. However, there are cases when even this representation is not sufficient. One example is the simplex method, where wrongly generated non-zeros can slow down the solution algorithm and can lead to false results. One possible way to reduce the chances of the occurrence of such events is implementations using absolute and relative tolerances in order to mitigate numerical errors. There are several open-source linear algebraic libraries like BLAZE, but they do not handle numerical errors. There are techniques that can greatly improve the accuracy of floating point additive arithmetic operations [1, 2, 3] but they are very slow. For example, sorting of the addends can increase the result's accuracy [4], but the drastic slow-down is unacceptable in many applications. Our aim is to develop an efficient linear algebraic library that supports increased accuracy by heuristics while the incurred slowdown factor is minimal. We can achieve

this by utilizing some future proof advanced features of Intel's SIMD processors. Of course, the real benefit in speed of these methods appears in computationally demanding applications like in Gaussian elimination, matrix multiplications. Particularly, since matrix by matrix multiplication needs dot-products, this operation can generate a significant amount of numerical error.

The paper is organized as follows. The behavior of numerical errors is shown in Section 2. Intel's SIMD architecture and the cache system are introduced in Section 3. We propose our simplified stable addition operation in Section 4, its SIMD implementation is also presented. The next section introduces our conditional branching free stable dot-product technique for the C programming language, and the SIMD based dot-product operation. In Section 6 the computational performance of the introduced implementations is compared and explained. Finally, in Section 7 we present our conclusions.

The introduced methods in this paper often use bit manipulation operations. One bit can be 1 or 0. The value of a bit can be changed by hardware and/or software. If a bit is changed to 1, we say that we *set* the bit. On the other hand, if we want to ensure that the bit is 0, we say the bit is *cleared*.

2 Numerical errors in scientific computations

The numbers used in scientific and engineering computations can be represented in several ways. A very reliable and accurate method is the symbolic number manipulation provided by Maple [5], MATLAB's Symbolic Math Toolbox [6], or Mathematica [7, 8]. However, there are applications, where these tools are not available. We are focusing on the floating point numbers [9, 10], which are easier to use, but they can have accuracy problems [11]. Floating point numbers (whether single or double precision) have three fields: Sign bit, significand and the exponent. The number of significand bits is fixed. The painful consequence of this principle is that the numbers with large magnitude have lower precision than smaller numbers. This can lead to the so called *rounding error*. Let a and b be nonnegative numbers with $a \gg b$. If we compute the value of $a+b$, it can happen that the result will be just a.

The other source of errors is *cancelation*, also known as *loss of significant digits*. If in theory $a = -b$, and $a, b \neq 0$, we expect that $a+b = 0$. However, if a and b may carry numerical inaccuracies the computed sum can be a small number ε which is usually computational garbage. If the execution of the program depends on the zero test of the result it can go in the wrong direction. Our primary interest is the numerical behavior of optimization software. In this area considerable efforts have been made to properly handle the emerging cancelation errors. Our stable vector operations also have been designed to handle this type of error in an efficient way.

3 Intel's SIMD architecture

The SIMD (Single Instruction, Multiple Data) architecture provides a tool to perform the same low level operations on multiple data in parallel [12]. It was successfully used in the simplex method [13], and in other numerical algorithms [14]. The

old Intel CPUs used a stack for storing 32, 64 or 80 bit floating point numbers. This architecture can perform the current operation only on a single data. In 1999 Intel introduced the SSE (Streaming SIMD Extensions) instruction set in the Pentium III processor. It contains 70 new instructions, which can operate on single precision numbers. The processor has 8 brand new, 128 bit wide registers, named XMM0, XMM1, ..., XMM7. One XMM register can store 4 single precision numbers. The arguments of the operations are the registers, and the result will be stored in such a register. For example, if we add the content of register XMM1 to XMM0, the CPU adds the first number in XMM0 to the first number of XMM1, and so on, as it is shown in Figure 1.

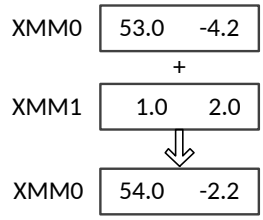


Figure 1
Addition using XMM registers

However, SSE does not support 64 bit floating point operations. Pentium 4 introduced SSE2, which supports 64 bit numbers as well. The XMM registers are still 128 bit wide, so one register can contain 2 double precision floating point numbers. SSE3 and SSE4 processors have added some expansion to the instruction set, like dot product and integer number support.

In 2011, Intel added the AVX (Advanced Vector Extensions) instruction set to the CPUs. This extension doubles the size of the XMM registers to 256 bit wide. It is called YMM. Now one register can store four 64 bit floating point numbers. Moreover, AVX's have 16 YMM registers. The AVX2 instruction set provides some integer operations with the new registers.

The modern Intel CPUs have one more useful feature. Namely, they have two memory ports. It means that, while the CPU calculates, they can load some other data from the memory in parallel.

Cache

In this section a brief summary of the CPU caching is given because it has some non-intuitive properties: The wrong utilization of the cache cannot achieve the highest performance. The communication between the CPU and the main memory is much slower than the speed of the CPU, i.e., while the CPU is waiting for the memory, it can execute a lot of instructions. To keep the CPU working engineers have designed cache memory between the CPU and the main memory. The cache uses faster circuit elements, but it is more expensive, so the cache size is limited relative to the main memory. Typical cache sizes are 3-10 Mbytes today, while the main memory can be

32 Gbytes. Moreover, modern CPUs have more cache levels, where the lower levels are smaller, but they are faster.

The memory is divided into so called cache lines, which are certain lengths of memory partitions. Typical lengths are 32 or 64 bytes. If an instruction reads some bytes from a given memory address, the total cache line that contains the required data moves to the cache. If later instructions load an adjacent memory address its content will already be in the faster cache, thus the reading time is reduced. However, as the cache size is limited, the CPU has to make room for a new cache line if the required data item is not in the cache. In this case a formerly used cache line is dropped out and its content is written back to the main memory if it is necessary. When an instruction writes to a given address, the corresponding cache line is loaded into the cache, and the instruction writes there. In this case, the content of that memory address has a copy in the cache, which is different. This cache line is called dirty, but if we write this content back to memory, this flag is cleared.

There is a little intelligence in the cache controller. If the CPU senses that the software accesses adjacent memory addresses it loads some next cache lines. So, if we read or write a memory region from its beginning to its end, the currently needed data will already be in the cache.

The SSE2 and AVX support bypassing the cache for memory writing. In this case the cache line of the current memory address is not loaded and the CPU writes into the main memory directly. We call this non-temporal writing. Obviously, this mode is much slower in itself. However, we can keep the more important data in the cache. What happens if we add two large vectors (larger than the cache), and the result is stored in a third vector? Without bypassing, the CPU reads the next two terms of the sum and it has to write the result to the memory. At first, the result is placed into the cache. However, since the vectors are too large, and their contents fill the cache, the CPU has to drop out an older cache line to the memory. If the cache line of the result is not prepared for the cache the CPU has to load that cache line and, obviously, drops out an older line too. The non-temporal writing prevents the CPU from loading the cache-line of the destination, so it drops out older cache lines if and only if there is no more room for the input data. Finally, the performance of this algorithm is improved.

4 Vector addition

In computational linear algebra (on which many optimization algorithms rely) vector addition is one of the most frequently used operations.

Let $\mathbf{a}$ and $\mathbf{b}$ be two n dimensional vectors, $\mathbf{a},\mathbf{b} \in \mathbb{R}^n$. We propose our implementations of the following vector addition operation:

$$\mathbf{a} := \mathbf{a} + \lambda \mathbf{b},$$

where $\lambda \in \mathbb{R}$. In detailed form:

Input: $\mathbf{a}, \mathbf{b}, \lambda$
Output: $\mathbf{a}$
1: **for** $i := 1$ to n
2: $\quad a_i := a_i + \lambda b_i$
3: **end**

Figure 2
Naive vector addition

If cancelation errors occur, the implementation shown in Figure 2 can generate fake non zeros. This error can be handled with an absolute tolerance ε_a. If the absolute value of the sum is smaller than the ε_a, we set the result to zero as in Figure 3.

Input: $\mathbf{a}, \mathbf{b}, \lambda$
Output: $\mathbf{a}$
1: **for** $i := 1$ to n
2: $\quad a_i := a_i + \lambda b_i$
3: $\quad$ **if** $|a_i| < \varepsilon_a$ **then**
4: $\quad\quad a_i := 0$
5: $\quad$ **end**
6: **end**

Figure 3
Vector addition using absolute tolerance

The absolute tolerance cannot adapt to the magnitudes of the input values. The solution can be the use of a relative tolerance ε_r. In 1968 William Orchard-Hays [15] suggested the following method using this tolerance: If the sum is much smaller relative to the largest absolute value of the input numbers the result is set to zero, see Figure 4.

Input: $\mathbf{a}, \mathbf{b}, \lambda$
Output: $\mathbf{a}$
1: **for** $i := 1$ to n
2: $\quad c := a_i + \lambda b_i$
3: $\quad$ **if** $\max\{|a_i|, |\lambda b_i|\}\varepsilon_r \geq |c|$ **then**
4: $\quad\quad c := 0$
5: $\quad$ **end**
6: $\quad a_i := c$
7: **end**

Figure 4
Vector addition using relative tolerance, Orchard-Hays's method

Determining the maximum of two numbers uses conditional branching. We propose a simplified method which uses fewer operations. It is sufficient to multiply the absolute value of one of the input numbers by the relative tolerance. In this way we can save an absolute value and a conditional branching step. The result can be

close to zero if the input values have the same order of magnitude and their signs are different. The useful value of the relative tolerance can be different from that of the Orchard-Hays method.

Input: $\mathbf{a}, \mathbf{b}, \lambda$
Output: **a**

```
1: for i := 1 to n
2:         c := a_i + λ b_i
3:         if |a_i| ε_r ≥ |c| then
4:                 c := 0
5:         end
6:         a_i := c
7: end
```

Figure 5
Vector addition using simplified relative tolerance

The implementation shown in Figure 3 requires one compare and a conditional jump instruction. Our simplified implementation with relative tolerance uses one addition, two multiplications, two assignments, two absolute values, one compare and one conditional jump. Orchard-Hays's implementation needs one more absolute value and a conditional branching. The additional operations cause overhead in time, so these implementations are slower than the naive one.

4.1 SIMD vector addition

Conditional jumping slows down the execution of the program because it breaks the pipeline mechanism of the CPU. So it is worthy to try to implement the algorithms in a way that avoids conditional jumps. Intel's SIMD architecture contains several instructions which help us design such an implementation. We will use the following instructions:

- *Move:* Moves the content of a register to another register.
- *Multiply:* Multiplies the number pairs of two registers.
- *Add:* Adds the number pairs of two registers.
- *And:* Performs a bitwise AND between two registers.
- *Compare:* Compares the number pairs of two registers. If they are identical the destination register will contain a bit pattern filled by 1's, otherwise 0.
- *Max:* Chooses the larger of two numbers stored in two registers. It is used for the implementation of Orchard-Hays's addition method.

The detailed description of these instructions can be found in [16]. The key point of the conditional jump aware implementations (called accelerated stable addition in this paper) is the compare instruction. It compares the number pairs and stores the results in a register. If the register contains two double pairs then the comparator puts two bit patterns in the destination area. One pattern can be filled by 1 if the

result of the comparison for the related number pair is true, otherwise 0 as it is shown in Figure 6. These bit patterns can be used for bit masking.

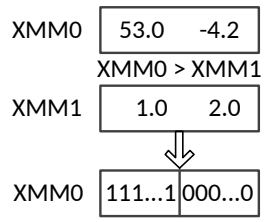


Figure 6
The compare instruction of the SSE2 instruction set

Figures 7 and 8 show the flowchart of the SSE2 implementations of our stable add operations with relative and absolute tolerances. The algorithms add two number pairs loaded to registers XMM0 and XMM1. The final result is placed in XMM2.

The implementations have two main phases: initialization, and process. We prepare some registers to store the value of λ (XMM7), ε_r (XMM4), ε_a (XMM6) and the absolute value mask (XMM5). In the process phase we perform the stable add operations for the successive number pairs, without modifying registers XMM4-XMM7. Figures 7 and 8 show only one iteration in the processing phase. One iteration of the absolute tolerance version stable adder performs 6 steps:

1. Multiply XMM1 and XMM7, store the result in XMM1, XMM1 will store λb_i.
2. Add XMM1 to XMM0, so XMM0 stores $c = a_i + \lambda b_i$.
3. Move XMM0 to XMM2. We have to store the original value of c, in order to use its absolute value in later steps.
4. Bitwise AND between XMM2 and XMM5, store the result in XMM2. Therefore XMM2 stores $|c|$.
5. Now we have to compare $|c|$ and ε_a. If $|c| < \varepsilon_a$, then the CPU sets the bits of the corresponding floating point number in XMM2, otherwise clears them.
6. Bitwise AND between XMM2 and XMM0. After this step, if $|c| < \varepsilon_a$ then XMM2 stores zero, because of the cleared bit mask in XMM0, otherwise XMM2 stores c.

The stable add operation that uses relative tolerance performs 9 steps in one iteration:

1. Multiply XMM1 and XMM7, store the result in XMM1, XMM1 will store λb_i.
2. Move XMM0 to XMM2. We have to store the original value of a_i and λb_i, in order to use their absolute value in the later steps.
3. Add XMM1 to XMM2, so XMM1 stores $c = a_i + \lambda b_i$.

4. Move XMM2 to XMM3, because we will use the absolute value of c in the next steps, but we will need the original value of c as well.
5. Bitwise AND between XMM3 and XMM5, store the result in XMM3. Therefore XMM3 stores $|c|$.
6. Bitwise AND between XMM0 and XMM5, XMM0 stores $|a_i|$.
7. Multiply XMM0 and XMM4, and store the result in XMM0, so XMM0 stores $|a_i|\varepsilon_r$.
8. Now we have to compare $|a_i|\varepsilon_r$ and $|c|$. If $|a_i|\varepsilon_r < |c|$, then the CPU sets the bits of the corresponding floating point number in XMM0, otherwise clears them.
9. Bitwise AND between XMM2 and XMM0. After this step, if $|a_i|\varepsilon_r \geq |c|$ then XMM2 stores zero, because of the cleared bit mask in XMM0.

Each operation above belongs to exactly one SSE2 or AVX instruction, so the reader can easily reproduce our results. These implementations use several additional operations on top of the one addition and multiplication, so they have an overhead compared to the naive implementation. They use some additional bit masking steps, because the Intel's SIMD instruction sets have no absolute value operations. However, we can obtain the absolute value of a floating point number by clearing the sign bit. Therefore, we have to apply a bit masking technique to get the absolute values, as in the steps 5-7, in relative tolerance adder, and step 4 in absolute tolerance adder.

However, SSE2 performs every instruction between two number pairs in parallel, so this overhead is not significant. Moreover, AVX can execute the instructions between 4 number pairs, consequently, the overhead will be even lower. In order to improve the speed of the algorithms, our implementations utilize the two memory ports mentioned in Section 3: While one number pair is being processed, the next pair is loaded to other unused registers, so the delay of memory operations is decreased. This technique is used in our dot-product implementations. In the future, AVX-512 processors will further increase the performance.

We modified the above relative tolerance adder procedure to implement Orchard-Hays's method. After step 6, two additional steps are inserted:

1. Bitwise AND between XMM1 and XMM5, XMM1 stores $|\lambda b_i|$.
2. Use MAX operation between XMM0 and XMM1, XMM0 stores $\max\{|a_i|, |\lambda b_i|\}$.

5 Vector dot-product

The dot-product between two n dimensional vectors $\mathbf{a}$ and $\mathbf{b}$ is defined as:

$$\mathbf{a}^T\mathbf{b} = \sum_{i=1}^{n} a_i b_i.$$

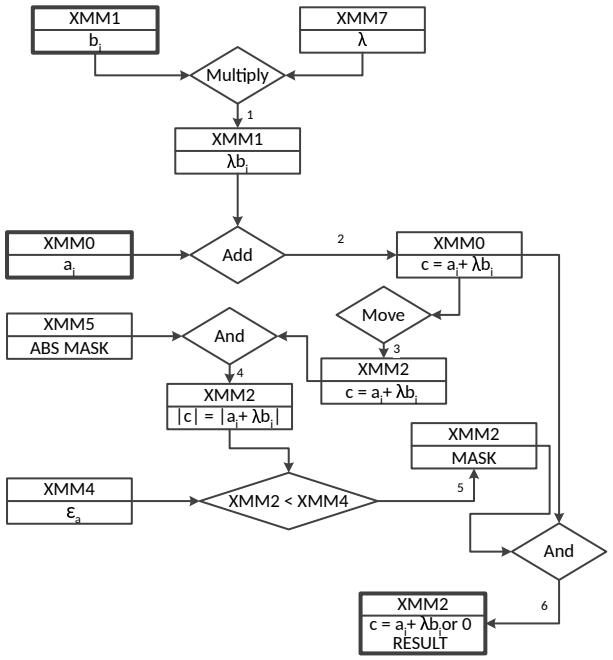


Figure 7
Flow chart of the stable add implementation, using absolute tolerance. Arrow numbers show the order of the operations.

Figure 9 shows the pseudo code of its naive implementation. The problem is that the add operation in line 3 can cause a cancelation error.

This error can be greatly reduced by using a *pos* and a *neg* auxiliary variables as introduced by Maros and Mészáros in 1995 [17]. Positive (negative) products accumulate in variable *pos* (*neg*). Finally, the result is the sum of *pos* and *neg* as shown in Figure 10. This final add is a stable add operation introduced in Section 4.

The conditional jump in line 5 breaks the pipeline mechanism and the execution slows down accordingly. We have developed a solution for C/C++ programs, where the conditional jump can be avoided and substituted by pointer arithmetic. This method can be used if the later introduced SIMD based methods are not available, for example the AVX is disabled by the operating system. The elements of an array are stored in adjacent memory addresses. If a pointer is increased by 1 in C/C++, it will refer to the next object. The most significant bit in the bit pattern of a double type variable stores the sign bit. If this bit is 1, the number is negative, otherwise it is positive. The conditional jump free implementation uses a double type array, where the first element stores the positive, the second one stores the negative sums. The current product is added to one of these elements. The address of the current sum variable is obtained by a C/C++ expression: The address of the array is shifted by the product's sign bit, as Figure 11 shows.

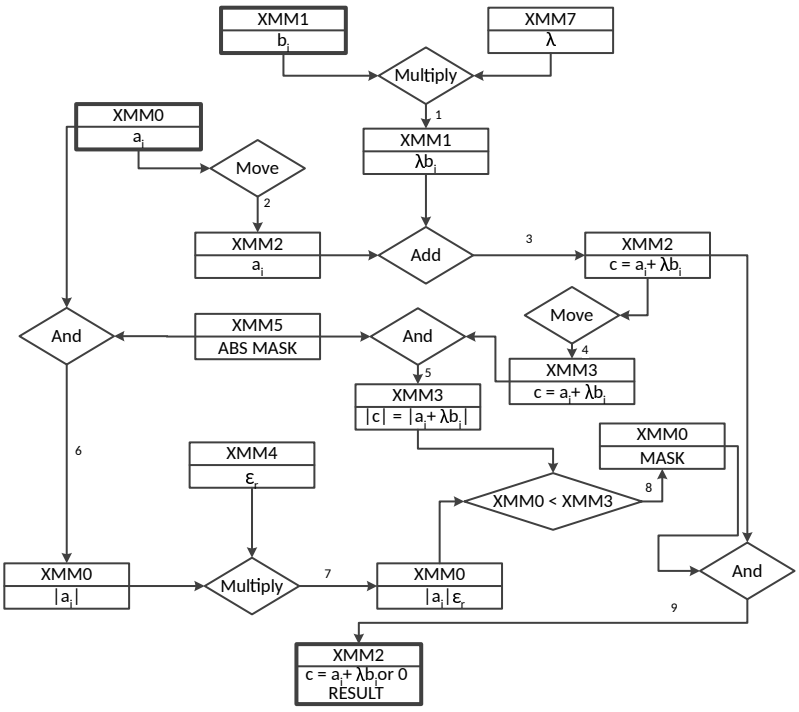


Figure 8
Flow chart of the stable add implementation, using relative tolerance. Arrow numbers show the order of the operations.

Input: $\mathbf{a}, \mathbf{b}$
Output: dp

1: $dp := 0$
2: **for** $i := 1$ to n
3: $\quad dp := dp + a_i b_i$
4: **end**

Figure 9
Naive dot-product

SIMD dot-product

The SIMD version of the dot product uses similar techniques introduced in Section 4.1. This implementation also has two phases, initialization and processing. We use XMM1 to store the negative products, XMM2 stores the positive products, and XMM4 contains zero for the comparison.

In the first step the product is loaded into XMM0. Of course, the multiplication can be supported by SSE2. The separation of positive and negative products can be implemented in 7 steps:

Input: $\mathbf{a}, \mathbf{b}$
Output: dp
1: $dp := 0$
2: $pos := 0$
3: $neg := 0$
4: **for** $i := 1$ to n
5: **if** $a_i b_i < 0$ **then**
6: $neg := neg + a_i b_i$
7: **else**
8: $pos := pos + a_i b_i$
9: **end**
10: **end**
11: $dp := StableAdd(pos, neg)$

Figure 10
Stable dot-product, where *StableAdd* is an implementation of the addition, which can use tolerances

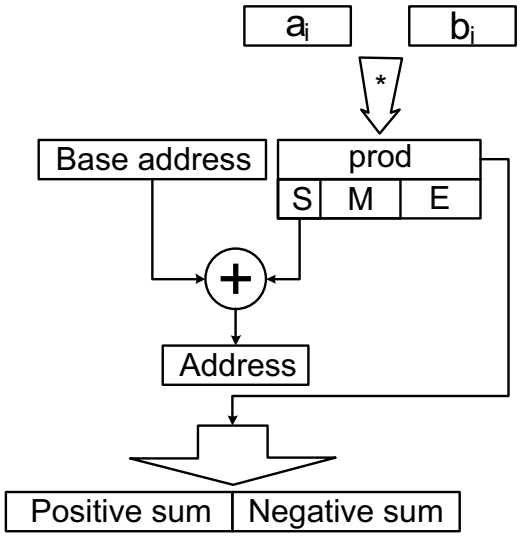
1. In order to keep the value of the product $a_i b_i$, we save the content of XMM0 to XMM5.
2. Move the content of XMM0 to XMM3, in order to perform the comparison between zero and the product.
3. Compare XMM3 with XMM4, if $a_i b_i < 0$, then the CPU sets the bits of the corresponding floating point number in XMM3, otherwise clears them.
4. Bitwise AND between XMM5 and XMM3. If $a_i b_i < 0$, then XMM5 stores $a_i b_i$, otherwise zero.
5. Add XMM5 to XMM1, i.e if $a_i b_i < 0$, then we add this negative value to XMM1, otherwise we add zero.
6. Bitwise AND between the inverse of XMM3 and XMM0. If $a_i b_i \geq 0$, then XMM3 stores $a_i b_i$, otherwise zero.
7. Add the content of XMM3 to XMM2, that is we update the positive sum.

Similarly to the SIMD accelerated vector addition, this dot product algorithm can be improved using AVX. The stable dot product uses fewer instructions than the stable add, so the performance of this implementation is better, as we will see in Section 6.

6 Computational experiments

In this section some benchmarking results are presented. The tests were performed on a computer with the following parameters:

- CPU: Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz
- Level 1 cache: 32 Kbyte
- Level 2 cache: 256 Kbyte




```
union Number {
 double num;
 unsigned long long int bits;
} number;
    double negpos[2] = {0.0, 0.0};
    [...]
    const double prod = a * b;
    number.num = prod;
    *(negpos + (number.bits >> 63)) += prod;
```

Figure 11
Handling positive and negative sums with pointer arithmetic without branching, where *S* is the sign bit, *M* is the significand and *E* is the exponent

- Level 3 cache: 3072 Kbyte
- Memory: 8 Gbyte
- Operating system: Debian 8, 64 bit
- Window manager: IceWM

The i5-3210M CPU has three cache levels. Intel processors have an inclusive cache architecture. It means that the higher level caches include the lower levels, so the test CPU has 3 Mbyte cache in total. Moreover, this CPU has two cores, where the L1 and L2 caches are unique in each core. However, the cores share the L3 cache, so it can happen that more than one process uses the L3 cache [18].

Our SSE2 and AVX implementations are written in assembly and compiled by NASM, version 2.11.08. In our C language implementations we used C++11 for

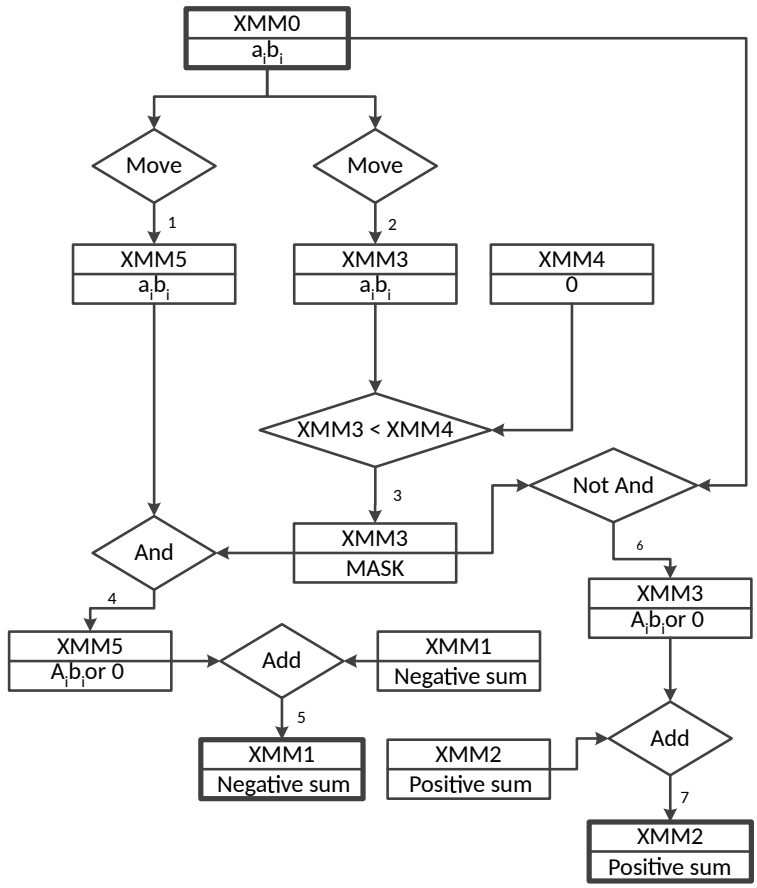


Figure 12
Flow chart of the stable dot product implementation. Arrow numbers show the order of the operations.

the benchmarking software, and the non-SIMD algorithm versions. The C++ compiler was gcc 4.9.2.

We have performed 80 measurements with different sizes of test vectors. In the sequel, s_i ($0 \leq i < 80$) denotes the size of one vector in the i^{th} test. In the first test, the size of a vector is 1000 elements ($s_0 = 1000$). The vector sizes grew exponentially: $s_i = \lfloor 1000 * 1.1^i \rfloor$, thus the largest vector size is 1862182 elements. Since one element is 8 bytes long, the smallest vector needs 8000 bytes, while the largest is 14.2 Mbytes long.

6.1 Vector addition

Each test was repeated 5000 times, and the execution time was measured. Based on the vector lengths and the execution time, the performance was calculated in number of FLOPS (Floating-point Operations Per Second). We have counted only the

effective floating point operations, i.e. the multiplication by λ and the addition. The number of effective floating point operations expresses how long input vectors can be processed by the current implementation. If an implementation uses additional auxiliary floating point operations (like multiplying by a ratio), that operations do not count.

The input vectors were randomly generated. If we add two numbers, then we have two cases: (1) The result is stable, so we keep it, (2) or the result violates a tolerance, so it is set to zero. Hence we have generated the input vectors in such a way that the likelihood for setting the result to zero is 1/2. This method moderately supports the efficiency of the CPU's branching prediction mechanism. Moreover, if it is required to set zero half of the results, it ensures that the non-vectorized implementations have to execute all of their branches.

We have to distinguish two cases of the vector addition operation:

1. $\mathbf{c} = \mathbf{a} + \lambda \mathbf{b}$, three vectors case
2. $\mathbf{a} := \mathbf{a} + \lambda \mathbf{b}$, two vectors case

where the memory areas of the vectors **a**, **b** and **c** are different. Since these cases use the memory in different ways we have tested them for every implementation.

6.1.1 Results for three vectors

If three different memory areas are used with cache, the cache is divided into 3 partitions, so the performance is decreased. However, if non-temporal memory writing is used, then larger vectors can be placed in the cache. Moreover, if the larger cache is still tight the non-temporal writing saves unnecessary memory operations. Therefore, this writing mode is recommended for large vectors. Figure 13 shows the results for the unstable implementations. It can be seen that the AVX is the best alternative, because it can perform four floating point operations per CPU cycle. The performance decreases if the vectors grow out of the available cache sizes. Since the L3 cache is shared among the cores, our process cannot use the whole cache, so the efficiency decreases sooner as the total vector sizes exceed the size of larger caches.

If the vectors are too large, the non-temporal SSE2 and AVX implementations have the same performance because they execute quick calculating operations, but the speed of memory operations is much slower than a floating-point operation. This holds for the cache writing implementations too, but their performance is the half of that of the non-temporal versions, because they use slower memory operations.

Figure 14 shows the results for the stable add implementations, where relative tolerance is used. Since more operations are used for one stable add step, the performance is lower than in the unstable case. If the vectors are larger then the non-temporal writing version with AVX is a little faster than the non-temporal SSE2 because the AVX instructions have to wait fewer times to read data from memory. While the 9 steps of the stable add are executed the CPU can read the next data into the cache.

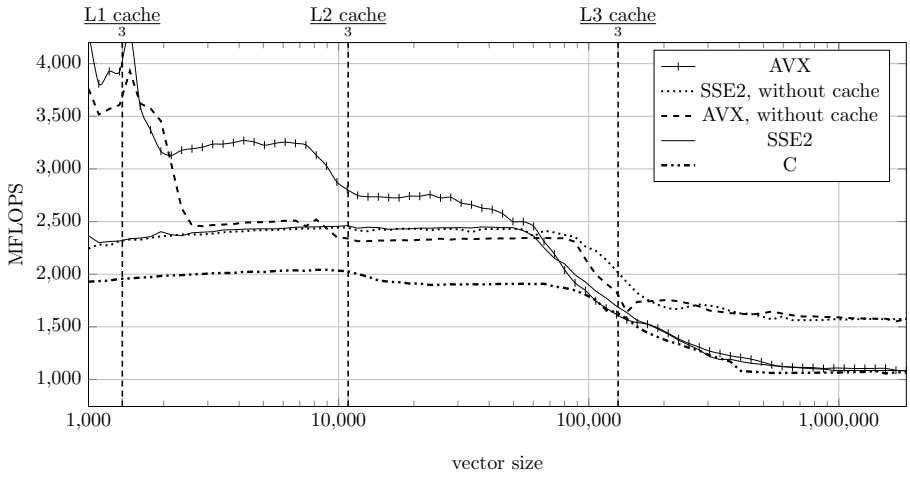


Figure 13
Performances of the unstable add vector implementations for three vectors

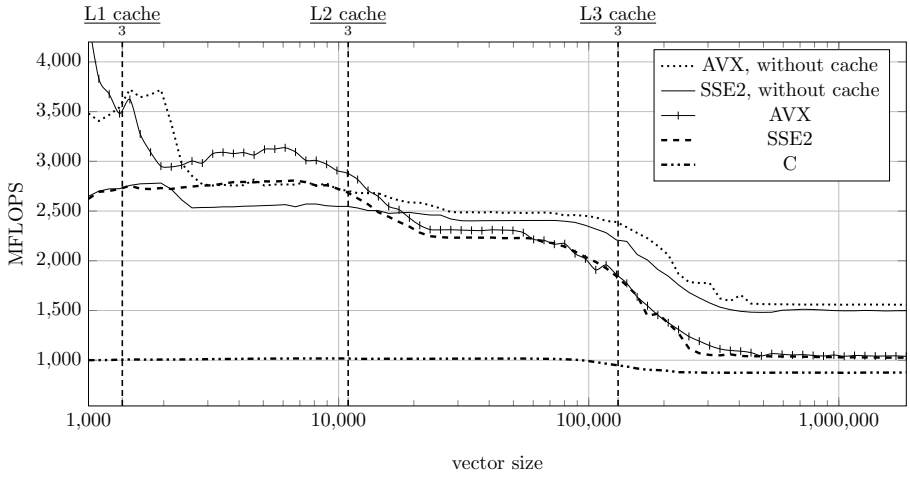


Figure 14
Performances of the stable add vector implementations, using relative tolerance, for three vectors

As Figure 15 shows, the performance of the absolute tolerance versions has a similar behavior to the unstable implementations but, of course, in this case the performance is lower.

6.1.2 Results for two vectors

If two vectors are used and one of them is the result the cache line of the current result memory area is in the cache. This involves that there is no additional communication between the cache and the memory, so the performance increases. Obviously, bypassing the cache is not unprofitable in this case, as Figures 16, 17,

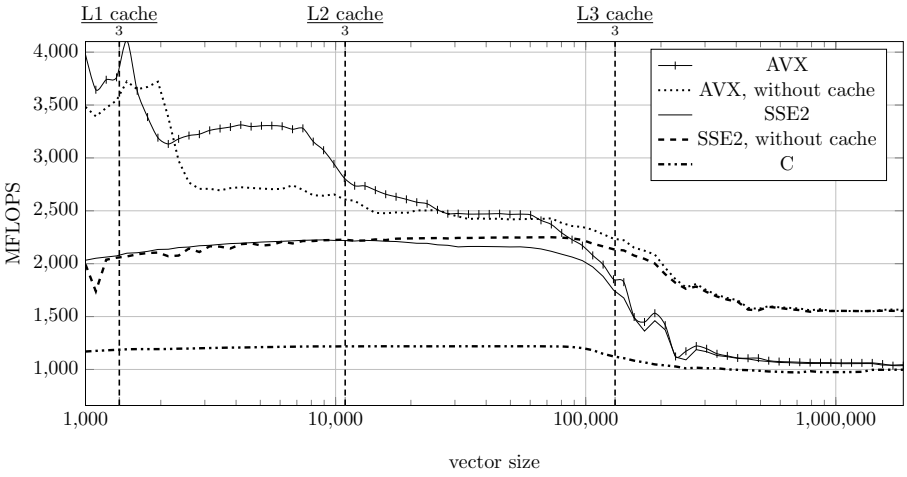


Figure 15
Performances of the stable add vector implementations, using absolute tolerance, for three vectors

and 18 show. If the cache is not bypassed, the overall performance is better than in the three vectors case.

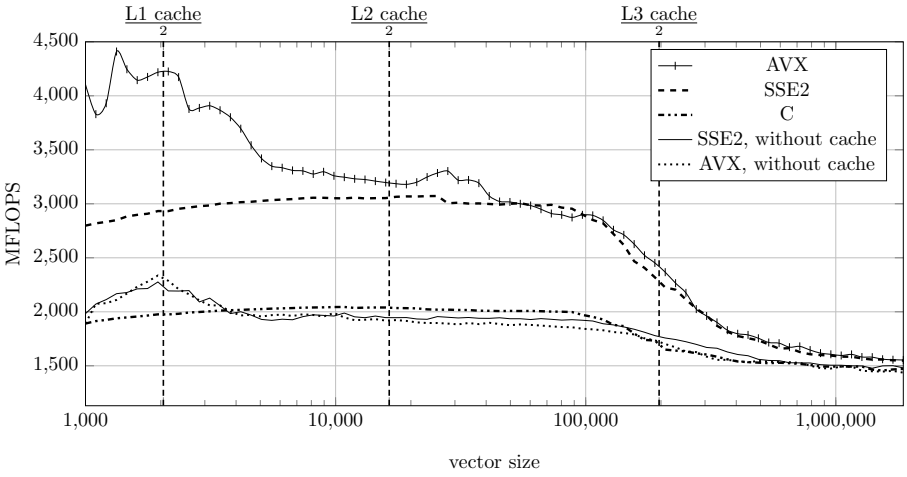


Figure 16
Performances of the unstable add vector implementations for two vectors

6.1.3 Orchard-Hays's relative tolerance method

Since SSE2 and AVX have a MAX operation which selects the maximum of two numbers, Orchard-Hays's relative tolerance test can be implemented on Intel's SIMD architecture. As mentioned in subsection 4.1 two additional operations are inserted into the assembly code; the max selector and an absolute value operation. The

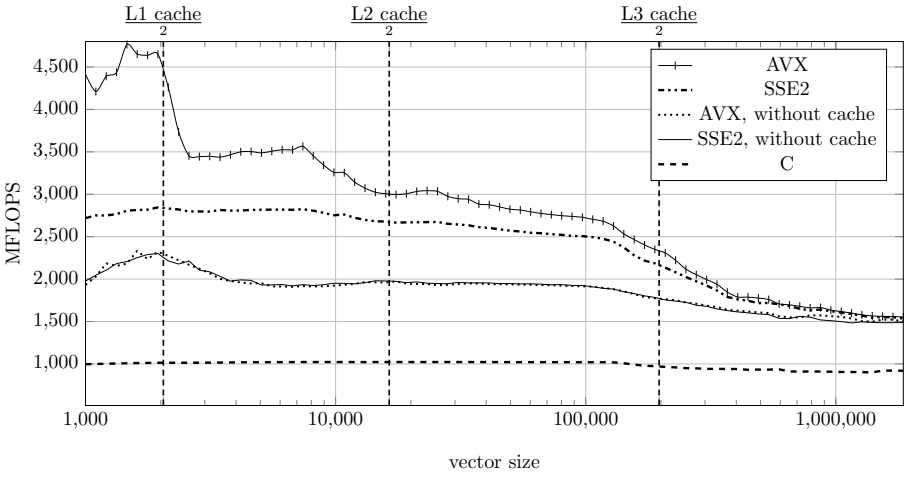


Figure 17
Performances of the stable add vector implementations, using relative tolerances for two vectors

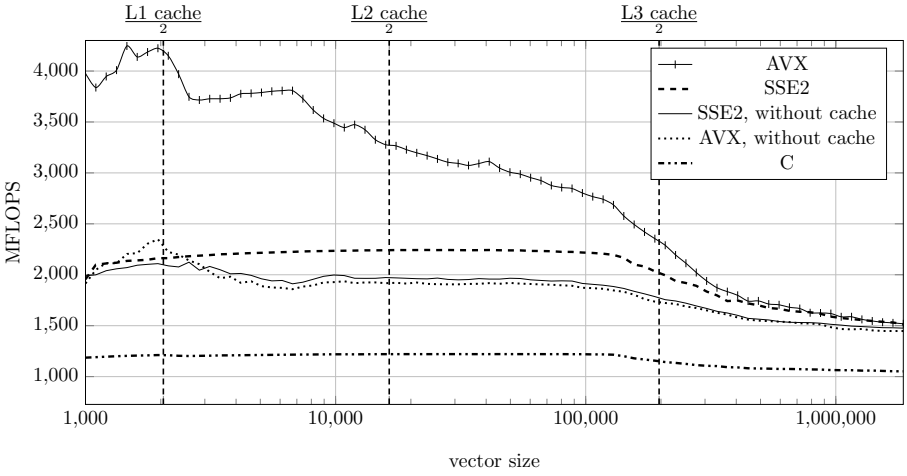


Figure 18
Performances of the stable add vector implementations, using absolute tolerance, for two vectors

modified implementation uses 11 instructions, where the max operation requires significant amount of execution time, as Figures 19-22 show. There were 800 measurement points that compare Orchard-Hays's and our method. In 607 cases, our algorithm is the fastest, the highest speedup ratio was 1.245 in the 3 vector SSE2 test, using cache. Our method behaved worse in the remaining 193 test points, the worst ratio was 0.905 in the 3 vector, AVX, and cache-free case. We mention that this is a very extreme case, in most of the cases, if our approach is worse, the ratio moves around 0.98. However, as we saw, a simple policy can be constructed: Depending on the vector's size, and numbers (2 or 3 vectors), we can choose between the cache and cache-free implementations. We can avoid most of the situations,

when our method's performance is lower than the original.

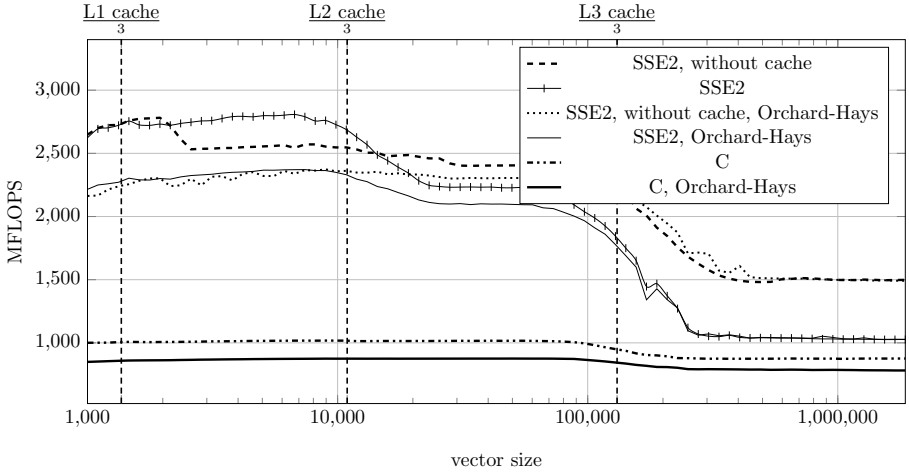


Figure 19
Performance comparison of our stable add implementations and the method of Orchard-Hays, with SSE2, using relative tolerance, for three vectors

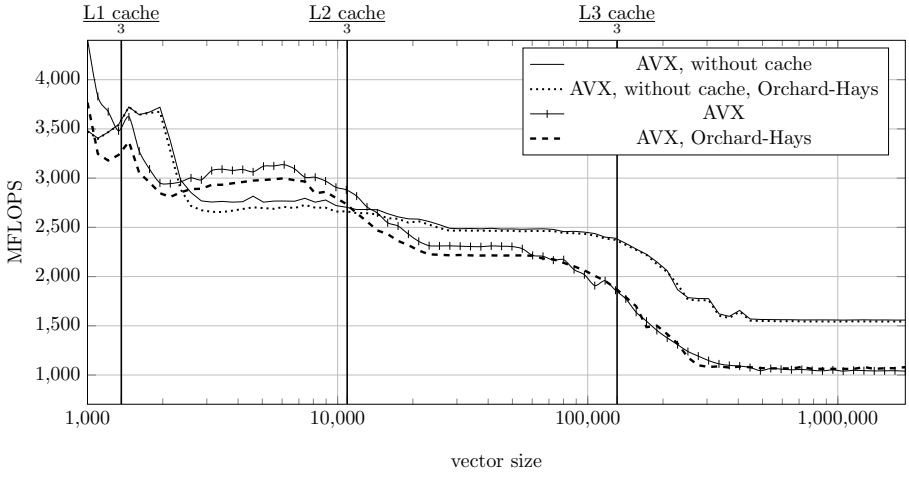


Figure 20
Performance comparison of our stable add implementations and the method of Orchard-Hays, with AVX, using relative tolerance, for three vectors

6.2 Dot-product

The dot product requires only two vectors and the result is a scalar value. Since, in general, the input vectors have much more than one element, writing time of the result to the memory is irrelevant. The stable AVX implementation uses only 7 instructions in addition to the loading, multiplying, and add operations, so its

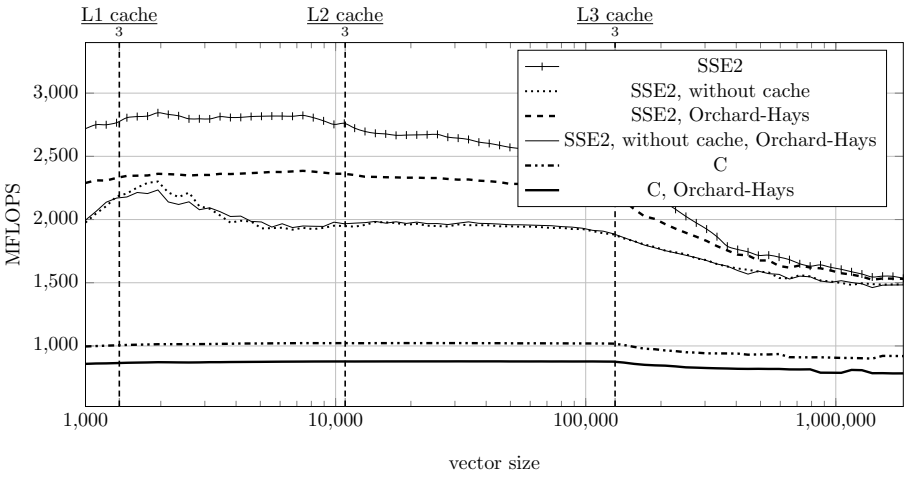


Figure 21
Performance comparison of our stable add implementations and the method of Orchard-Hays, with SSE2, using relative tolerance, for two vectors

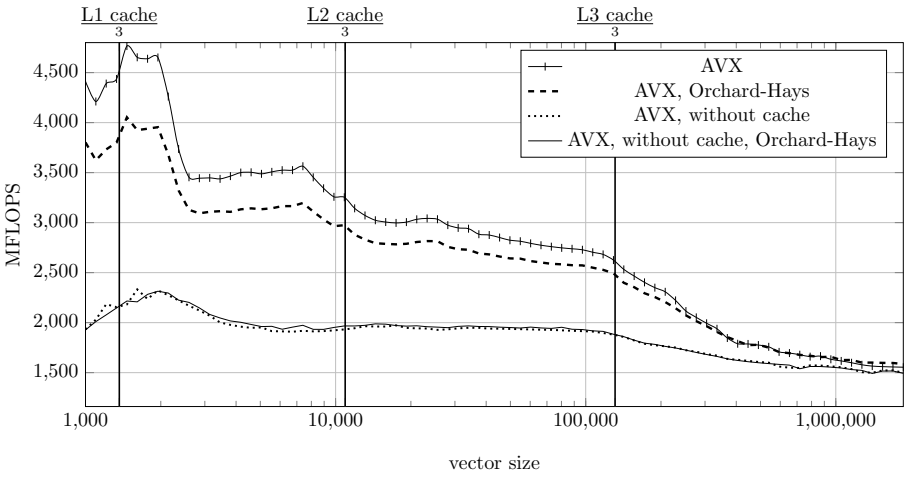


Figure 22
Performance comparison of our stable add implementations and the method of Orchard-Hays, with AVX, using relative tolerance, for two vectors

performance is better than the stable add. As Figure 23 shows, the performance of stable AVX dot product is close to the unstable AVX version. The stable SSE2 requires more cycles, so the performance is considerably lower than the unstable SSE2 version. The figure shows that if there is no SIMD support, the branching-free techniques can be very useful if the input vectors are sufficiently large.

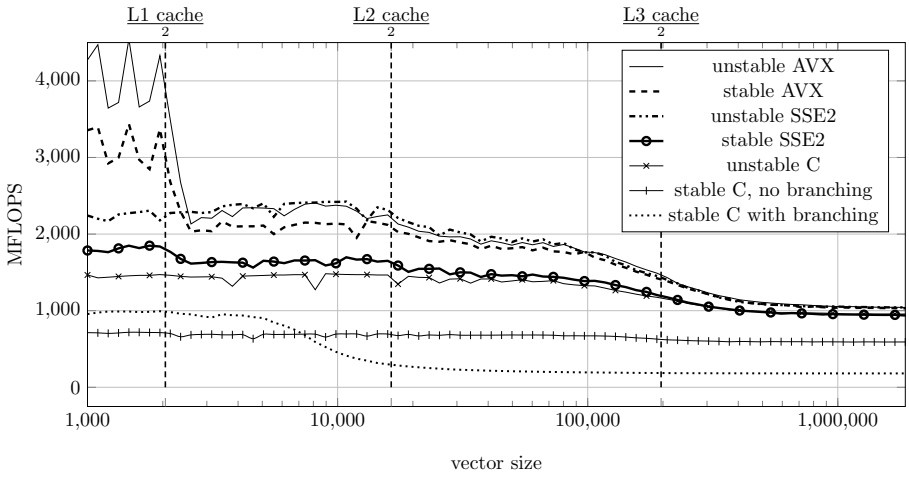


Figure 23
Performances of the dot product implementations

7 Conclusions

As the performance tests prove, our simplified stable add method is faster than Orchard-Hays's method. The applicability of our method is also tested by our simplex method implementation; the test problems of NETLIB were successfully solved. It is clear that our pointer arithmetic based stable dot-product implementation is much more efficient than the conditional branching version if the input vectors are sufficiently large. Moreover, the tests show that using Intel's SIMD instruction sets provides strong tools in order to implement the stable algorithms in an efficient way.

Modern Intel CPUs have at least two memory ports. So, while the next data set is loading from the memory, the CPU can execute complex computations on the previous set. This is why the AVX is so efficient in high performance stable computations.

References

[1] Ogita, T., Rump, S. M., and Oishi, S. (2005) Accurate sum and dot product. *SIAM J. Sci. Comput.*, **26**, 1955–1988.

[2] Langou, J., Langou, J., Luszczek, P., Kurzak, J., Buttari, A., and Dongarra, J. (2006) Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, New York, NY, USA SC '06. ACM.

[3] Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., and Zimmermann, P. (2007) Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, **33**.

[4] Dekker, T. J. (1971) A floating-point technique for extending the available precision. *Numer. Math.*, **18**, 224–242.

[5] Waterloo Maple Inc. Maple.

[6] MathWorks, I. (2005) *Symbolic Math Toolbox for Use with MATLAB: User's Guide*. MathWorks, Incorporated.

[7] Wolfram Research Inc. Mathematica.

[8] Wolfram, S. (1999) *The Mathematica Book (4th Edition)*. Cambridge University Press, New York, NY, USA.

[9] Knuth, D. E. (1997) *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[10] Muller, J.-M., Brisebarre, N., de Dinechin, F., Jeannerod, C.-P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., and Torres, S. (2009) *Handbook of Floating-Point Arithmetic*, 1st edition. Birkhäuser Basel.

[11] Higham, N. J. (2002) *Accuracy and Stability of Numerical Algorithms*, second edition. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[12] Hassaballah, M., Omran, S., and Mahdy, Y. B. (2008) A Review of SIMD Multimedia Extensions and their Usage in Scientific and Engineering Applications. *Comput. J.*, **51**, 630–649.

[13] Thomadakis, M. E. and Liu, J. (1997) An Efficient Steepest-Edge Simplex Algorithm for SIMD Computers. Technical report., College Station, TX, USA.

[14] Takahashi, A., Soliman, M. I., and Sedukhin, S. (2003) Parallel LU-decomposition on Pentium Streaming SIMD Extensions. In Veidenbaum, A. V., Joe, K., Amano, H., and Aiso, H. (eds.), *ISHPC*, October, Lecture Notes in Computer Science, **2858**, pp. 423–430. Springer.

[15] Orchard-Hays, W. (1968) *Advanced linear-programming computing techniques*. McGraw-Hill, New York.

[16] Intel (2013) *Intel 64 and IA-32 Architectures Software Developers Manual - Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z.*

[17] Maros, I. and Mészáros, C. (1995) A numerically exact implementation of the simplex method. *Annals of Operations Research*, **58**, 1–17.

[18] Intel (2013) *Intel 64 and IA-32 Architectures Software Developers Manual - Volume 3A: System Programming Guide, Part 1.*