Bachelor thesis at the Computer Science Institute of the Freie Universität Berlin

Computer Systems & Telematics workgroup

Performance optimization of real-world algorithms on modern processor architectures

Malte Rohde
malte.rohde@inf.fu-berlin.de

Supervised by: Prof. Dr. Marcel Kyas and Dipl.-Inform. Heiko Will

May 4, 2014

Abstract

While clock rates of modern general purpose processors seem to have reached their peak at around 4 Gigahertz, extensions to the instruction set such as SSE and AVX have become more and more important for performance critical applications. Most recent compilers make use of these instructions automatically when told to optimize an application for speed. Still, it sometimes may be necessary to manually take advantage of CPU features that the compiler is not capable of using in the very situation. In this bachelor thesis I am exploring ways to optimize performance of real-world algorithms using techniques such as parallelization and manual cache organization. Based on the example of a lateration algorithm simulator it will be shown how compiler intrinsics and code restructuring can be used to measurably improve execution performance.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, 3.4.2012

Malte Rohde

Contents

1 Introduction

In the early days of general-purpose computing, processing power was a scarce resource. Expensive, room-filling mainframes were often shared among numerous persons and thus their utilization needed to be as efficient and time-saving as possible. Performance, in terms of both minimum processor cycles and memory comsumption, was a critical feature for any software written in those days. Then, with the emergence of the personal computer and fast general-purpose microprocessors, application performance became far less important and programmers began to pay less attention to it when designing their applications. Today, small computer towers sitting below everyone's desk outperform all of those early mainframes and provide more than enough processing power for most everyday uses. When it comes to creating usual desktop applications or low-scale web applications, programmers can treat both memory and CPU power as effectively abundant resources. However, there are still plenty of cases where software performance matters to the user or is even a critical property of the application. For instance, in multimedia applications, video games, or embedded systems good performance is needed to create a pleasant user experience. In scientific computing or industry control systems, bad software performance often is intolerable.

Today, computer science lecturers tend to reduce algorithm performance to asymptotic complexity. An algorithm's performance is usually classified using the *Big-O notation*, which indicates an upper bound on the growth of the number of computational "steps" that the algorithm performs relative to the input size (referred to as N), while it neglects the constant amount of time needed for each step. This concept is based on the assumption that the performance impact of the so-called *constant factor* decreases with growing N, up to the point where performance is entirely dominated by the size of the input value. The Big-O notation has proved especially valuable for comparison between algorithms, as the constant factor depends on the internals of the system executing the algorithm as well as implementation details and therefore is hard to measure. By contrast, identifying the asymptotic complexity of an algorithm is merely a matter of counting the depth of loop nestings in its implementation. However, when it comes to real-world applications, complexity theory often does not tell much about the application's performance. In real-world applications, where the input size may be small, the constant factor often plays an at least equally important role.

Sorting algorithms, a popular research topic for decades, present a very good example. Consider the well-known sorting algorithms *Quicksort* and *Heapsort*[1]: Quicksort, on the one hand, has a best-case complexity of $O(N \cdot \log N)$ and a worst-case complexity of $O(N^2)$, whereas, on the other hand, Heapsort provides a guaranteed complexity of $O(N \cdot \log N)$ in all cases. Still, in most everyday use cases, especially when N is little, the smaller constant factor of Quicksort outweighs Heapsort's superior asymptotic performance. This is due to the fact that a "step" of the Quicksort algorithm consists of only a single comparison operation and a few data movements, whereas Heapsort requires a multitude of comparisons and moves in order to maintain the underlying data structure, a binary heap. As a consequence, Heapsort is rarely implemented.

Having emphasized the importance of the constant factor, it should have become clear that, apart from choosing the right algorithm to solve a particular problem, implementation details are crucial for a real-world application's performance. Therefore, the knowledge of various software optimization techniques remains a vital part of every programmer's profile. Knowing how the CPU, its caches, and the memory work internally helps to understand where to find bottlenecks and how to avoid them. Some optimization techniques are mere guidelines that should be remembered and considered whenever writing a piece of software, whereas others are rather advanced measures only useful in rare and specific situations. In general, when optimizing an application,

[1] For reference, consult [Knu98, pp. 113–122, 144–148].

the programmer's main goal should be to use all system components to their maximum capacity and to prevent any of them from stalling. For example, knowing about the various execution units embedded into the CPU, the programmer should try to use them all simultaneously, rather than using one after another. Similarly, he should keep both the CPU and the memory controller busy at all times, trying to prevent them from waiting for one another. The programmer should particularly keep the CPU's caching mechanism in mind, as it is ntended to improve performance rather than hinder it (e.g., due to *cache misses*, see Section 3.1.2), yet it requires the application to be designed in a cache-friendly way. Most performance bottlenecks in an application result from the programmer's failure to utilize the capabilities of the system components to the fullest. Lately, with processor speeds apparently having reached their limits at around 3-4 GHz, parallelization techniques have been brought into focus by CPU manufacturers and researchers alike. Both Intel and AMD have started to primarily ship multi-core microprocessors in their end user product segments. These processors' superior performance relies on the programmer producing highly parallelized concurrent applications. Apart from that, most modern x86 descendants feature extended instruction sets that allow data-parallellism. These SIMD (single instruction, multiple data) instructions enable the programmer to issue only one instruction to process multiple data values at a time. Introduced by Intel for the Pentium III in 1999 and later adopted by AMD, the *Streaming SIMD Extensions* (SSE) have become the most advanced and wide-spread SIMD technology available. While it was originally intended to support multimedia processing and thus mainly contained multimedia specific instructions, it matured over time and became an almost fully-fledged vector processing unit. Processors implementing the latest SSE incarnation, SSE4, feature 16 *media registers* called `xmm0` to `xmm15`, each of them begin 128 bits wide, and numerous instructions for parallel floating point and integer calculations. For example, using these, the programmer can choose to calculate the square root of four 32 bit floating point values within a single instruction, which would take about the same amount of time as its scalar counterpart[1]. However, using SIMD extensions for optimizational work falls into the second category of optimization techniques mentioned above. These instructions only turn to account in situations where one has highly parallelized applications preferably consisting of mostly calculation intensive code.

In the following, I will show how implementing various optimization techniques can greatly improve performance of a real-world application written in C. The LS^2 simulation engine written by Heiko Will, Thomas Hillebrandt, and Marcel Kyas is a high performant algorithm simulation framework created to evaluate the accuracy of various lateration algorithms. Although the application already features highly optimized C code, average runtimes of some of the implemented algorithms are still intolerably lengthy. For further optimization, I will mainly make use of SSE instructions for parallelized floating point operations and elaborate on the pitfalls that arise from it. Afterwards, I will assess the optimization measures' success using empiric benchmarks, which I will try to design to be statistically reliable. I will additionally describe the approach taken for development, which may provide some insight into what distinguishes optimizational from implementation work.

The main goal of my work is to significantly enhance the performance of three of the algorithms included in LS^2, namely *Geolateration*, *Voting Based Location Estimation*, and *Adapted Multilateration*. The main part of this thesis (Section 4) will therefore be comprised of a detailed depiction and analysis of this case study. As a secondary goal, I would like this thesis to become generally useful as a guide towards SSE programming, although the extent of this will certainly be somewhat limited. Nevertheless, in Section 3 I will present an overview of software perfor-

[1] Actually, the `FSQRT` instruction of the x87 FPU may even be a bit slower than the SSE version `SQRTSS/PS` (at least on an Intel "Wolfdale" CPU), yet provides for a greater precision as it internally operates on 80 bits. See [Fog11a] for instruction benchmarks.

mance and optimization theory. Therein the reader will find some general information on modern processor architectures and how to make use of their sophisticated internals. The bulk part of Section 3 will describe special techniques for code vectorization, for example, how to transform conditional jumps of scalar code into logical functions and assignments that can be deployed in SSE code.
Research done for this thesis concentrated on modern Intel processor architectures and the GNU compiler collection (gcc) and may not apply to other architectures or compilers. Although, since SSE is an industry standard that all major x86 vendors have agreed on and that is also well supported by all popular compilers, it is very likely that at least the basic concepts can be transferred to other processors and compilers. As a final remark, please note that there is not a "single way" in software optimization and the information presented in this thesis should not be interpreted as a complete summary of the topic.

2 Related work

Related work of this thesis (in the narrow sense of the word, that is research about how to apply optimization techniques and SIMD programming to existing real-world applications) is hard to find. Tuomas Tonteri discussed the optimization of some scientific algorithms in [Ton12]. He explains in detail how SSE can be used to speed up N-particle dynamics simulation and ray sphere intersection testing, among other algorithms. Cort Stratton provided a case study on a SSE-optimized matrix-vector multiplier in [Str02]. In this article he gives a step-by-step report on how he iteratively improved the multiplication algorithm and in doing so explains the CPU provided levers for optimization such as instruction pairing and data prefetching. Intel engineer Guy Ben Haim et al. wrote an article [HZS09] about SSE optimization of an image processing algorithm. This very detailed research gives some valuable insight on SSE-friendly characteristics of algorithms such as data layout and inherent parallelism.
Regarding instructional material about software optimization and specifically SSE utilization, one can find a wealth of works on the internet. Most notably, Danish researcher Agner Fog wrote five excellent books about various aspects of software optimization that he continuously updates and publishes on his website[1]. These include a guide for high-level optimization with C++ and SSE intrinsics [Fog11b] as well as a book [Fog11a] featuring exhaustive tables of instruction latencies and throughputs for almost all current CPU models, the clock cycles measured by Fog himself. The former will represent the foundation of the next section of this thesis.
Likewise comprehensive and well-structured is "What every programmer should know about memory" [Dre07] written by Ulrich Drepper. In this article the author examines in depth the technical details of random-access memory and presents ways for the developer to optimize his applications based on these specifics.
Intel produced their own optimization manual [Int11], which is very extensive, yet rather low-level and focussed on assembler language. Still, this is the most comprehensive source for information about how to completely exhaust Intel CPUs and understanding the assembler examples is a helpful preparation for doing high-level optimization. Apart from this, Intel offers lots of well-written articles through its *Software Network*[2], although some of them are clearly targeted at compiler and low-level programmers.
Catalog-style information about x86 processors (e.g. instruction listings) can be found at sandpile.org[3]. Reference documentation on SSE compiler intrinsics is provided by Intel in their

[1] `http://www.agner.org/optimize/`, last accessed: May 4, 2014
[2] `http://software.intel.com/en-us/articles/`, last accessed: May 4, 2014
[3] `http://sandpile.org`, last accessed: May 4, 2014

extensive "Intel 64 and IA-32 Architectures Software Developer Manual" [Int12]. Additionally, Intel distributes a handy desktop utility called "Intel Intrinsics Guide"[1], which offers a browse and search interface for intrinsics and is especially helpful when working offline. Intrinsics are also well-documented at Microsoft's MSDN[2].

3 Optimization theory

The following section describes the basic theory of software optimization. First, I will summarize guidelines that should always be followed when optimizing a piece of software. Naturally, these will only be a selection of the advice provided by relevant manuals. Then, I will go into detail when explaining concepts of using the SSE instruction set. The section is completed by a small remark on compiler optimization. While the advice given in this section is focussed on C/C++ code, most of it should be valid for other programming languages as well.

3.1 Basic principles

The first lesson a programmer learns when looking into software optimization is to *avoid premature optimization*. This advice is based on a statement given by Donald E. Knuth in [Knu74], "We *should* forget about small efficiencies, say about 97 % of the time: premature optimization is the root of all evil". Knuth continues to say that in the remaining (critical) 3 % of the code the programmer should look carefully for optimization opportunities, "but only *after* that code has been identified". The common notion of Knuth's words is that, while writing a piece of software, the programmer should not care about performance, until he can guarantee the code's correctness. Afterwards, he should find the most critical parts in the code and concentrate optimization efforts on those particular parts. This opinion implies two valid points: First, optimized code will always reduce readability and increase complexity. This may lead to errors and code that is hard to maintain. Second, most of the time optimization is only worth its costs in the most time-critical parts of a software. However, Knuth supposedly did not mean that the programmer should not think about performance at all. In fact, some basic rules can be kept in mind along the way.
In general, the performance of an application is either limited by processor speed or by memory bandwidth. Knowing which one will be the limiting factor in a particular piece of code can be helpful to avoid certain bottlenecks from the beginning. In order to design the code in a resource saving manner, it is critical for the programmer to understand the internals of these core system components. In the following I will present some information on modern (x86) microprocessor features and on how to write code that makes efficient use of both the processor and the memory. However, since this is not the main topic of my thesis and there is plenty of information on the Web, I will single out only a few major items.

3.1.1 CPU performance

On modern processor architectures, the execution of an instruction is divided into several stages which are executed sequentially. To reiterate the textbook example (for example, see [MH99, p. 411], consider the following model consisting of 4 stages: First, the instruction needs to be fetched (instruction fetch phase, IF) from memory or from a code cache. Second, the instruction needs to be decoded (ID) and, third, executed (EX). At stage 4 the results of the instruction need to be written back to memory or registers (WB). When the instruction fetch unit finishes

[1] Downloadable at `http://software.intel.com/en-us/avx/`, last accessed: May 4, 2014
[2] `http://msdn.microsoft.com/en-us/library/26td21ds%28v=vs.80%29.aspx`, last accessed: May 4, 2014

its work, it immediately starts fetching the next instruction from the code, the decoder starts decoding it and so on. In other words, the instructions are processed in parallel. This staging of instructions and their parallel execution is called *pipelining*. While in theory this model can complete the execution of one instruction every cycle, in practise it if often limited by *data dependencies* between the instructions. If an instruction depends on the results of a preceding instruction, it needs to wait for the results to be written back to registers, effectively stalling the processor. See Figure 1 for an example: Here, the second instruction depends on the result of the first and thus its execution has to be delayed. Note that in this example the EX stage requires only one clock cycle, which is only true for simple instructions such as addition or subtraction of integers.

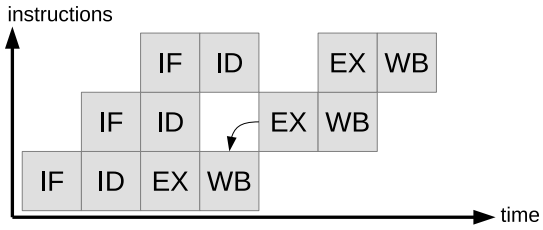


Figure 1: Model of a filled processor pipeline with a data dependency

As most modern microprocessors have multiple execution units for various types of operations (e.g. the x87 floating point unit that's sole purpose is to process floating point calculations), or even have multiple units of the same type, instructions that are executed on different execution unit can be processed in parallel. This is known as *out of order execution*, as the instructions do not necessarily need the same amount of clock cycles and a particular instruction may be outpaced by successive faster instructions, i.e., the instructions are executed "out of order". Some modern processors may even reorder instructions on the same execution unit. Out of order execution, just like pipelining, suffers from data dependencies.
In general, data dependencies only measurably diminish performance when appearing in longer sequences (called *dependency chains*), for example in loops. To minimize pipeline stalling, the programmer needs to reduce the number of data dependencies in such loops. Listing 1 displays an example taken from Agner Fog's optimization manual [Fog11b, p. 103]. The increment operation on the loop variable `i` and the floating point operations are executed on different execution units and thus are likely to be executed out of order. Therefore, the bottleneck is the more expensive floating point addition. In version 1, every floating point addition depends on the result (`sum`) of the preceding addition, which stalls the pipeline. In constrast, in version 2, two succeeding additions do not depend on each other and thus do not cause a pipeline stall. The technique demonstrated here is known as *loop unrolling*. It usually comes at the expense of additional instructions at the end of the loop (here: `sum1 += sum2;`), which can almost always be neglected.

Listing 1: Loop unrolling example

```
1  const int size = 100;
2  float list[size]; int i;
3
4  // Version 1:
5  float sum = 0;
6  for (i = 0; i < size; i++)
7    sum += list[i];
8
9  // Version 2:
10 float sum1 = 0, sum2 = 0;
11 for (i = 0; i < size; i += 2) {
12   sum1 += list[i];
13   sum2 += list[i+1];
14 }
15 sum1 += sum2;
```

Another positive effect of loop unrolling is that the loop control branch is evaluated less often. This often leads to better branch prediction results resulting in fewer mispredicted conditional jumps (see Section 3.3.2 for further explanation of conditional branches). Apart from loop unrolling, another simple technique to avoid data dependencies and to make full use of out of order execution is known as *instruction reordering* or *instruction pairing*. Simply put, instructions that do not depend on each other should be grouped whereas instructions having dependencies between them should be interleaved with other unrelated instructions. For out of order execution it is also desirable to mix instructions which are using different execution units, for example integer and floating point operations. It may not always be easily visible if such measures truly affect the processing inside the CPU, since modern CPUs generally have sophisticated instruction scheduling features and thus will often "outsmart" the programmer. Additionally, performance impact may vary across different CPU generations, for example when newer CPUs contain further execution units. Hence, it is necessary to thoroughly benchmark any optimized code and decide whether these measures are worth the loss of readability, which can be especially bad in the case of instruction reordering.

3.1.2 Memory performance

Stack vs. heap storage of variables. The memory of an application is divided into two major parts: The *stack* and the *heap*. The stack resides at the beginning of the memory block and is used in a last-in-first-out fashion. When an application allocates a variable on the stack (i.e., all local variables in C), it simply needs to move up the *stack pointer*. The heap usually resides at the end of the memory, growing down. It is used for dynamic memory allocation. When the application wants to allocate memory on the heap, for example because it wants to allocate a dynamically sized array, it needs to ask a heap manager for it (`malloc` in C). This manager will then look for a memory range that provides sufficient free space for the variable. Managing the heap is much more expensive compared to the simple stack approach, yet it may sometimes be impossible to store all variables on the stack. For instance, in other programming languages such as C89 or C++, it is not allowed to allocate memory on the stack when the size is not known at compile time. Apart from that, the stack is much smaller than the heap and some objects or large arrays may exceed its space limits. However, as the dynamic allocation overhead and additionally the potential memory fragmentation resulting from it reduce performance perceivably, frequently used variables should be stored on the stack whenever possible [Fog11b, p. 90].

Cache misses. Modern microprocessor architectures feature a multi-level hierarchy of caches which is used to speed up repeated accesses to data values. The Level 1 cache, being the (physically) closest cache available to the CPU, provides the fastest access time. With every successive cache (L2/L3), both access times and cache size increase. For simplicity, I will assume a model of only one cache in the following. Whenever an application uses or manipulates a variable, the CPU checks whether the memory segment where it resides is already loaded into the cache. If it is not, a *cache miss* occurs, resulting in the segment being fetched from main memory. Since the RAM is extremely slow compared to the CPU cache, the latter is a very time-consuming operation[1]. The programmer's goal is to reduce the number of cache misses produced by his code. The main motivation for today's cache architectures is the so-called *locality of reference* or *data locality*. Basically, data locality refers to probability observations on data access patterns, that are commonly divided into two types: First, temporal locality means that data that has been accessed recently is very likely to be accessed again soon. Second, spatial locality means that data that is spatially close to recently accessed data with regards to its position in the memory (i.e., the memory addresses are similar) is also likely to be accessed soon. Therefore it makes sense to not only cache a single variable that was recently used, but the whole memory range where this variable resides. To support the CPU's caching mechanism, the programmer should adapt his access patterns to these probability estimations. As Agner Fog puts it, "Variables that are used together should be stored together" [Fog11b, p. 88]. This boils down to simple strategies such as always allocating variables when one needs them. As an often used example for cache-friendly variable access, consider the array manipulation in Listing 2.

Listing 2: Sequential vs. non-sequential array access

```
1 int buf[1024 * 1024];
2
3 for (int i = 0; i < 1024; i++)
4   for (int j = 0; j < 1024; j++)
5     buf[i * 1024 + j]++;
6
7 for (int i = 0; i < 1024; i++)
8   for (int j = 0; j < 1024; j++)
9     buf[j * 1024 + i]++;
```

While both nested loops do the same thing, namely increment each element in the buffer, the crucial difference between them is the way the array offset is calculated. While the first loop walks through the array sequentially with the index always growing by one, the second loop "jumps" through the array in steps of 1024. The second approach results in a lot more cache misses, as the cache lines get repeatedly displaced by others.

3.2 Parallelization using Streaming SIMD Extensions

Created to meet the growing demand for fast multimedia functions in 1996, Intel's MMX technology was the first widely used instruction set extension to deploy the SIMD architecture to modern desktop processors. It used the lower 64 bit of the 80 bit x87 floating point registers to allow for vectorized calculation of 8 bit to 64 bit integers using special SIMD instructions. Its successor, the Streaming SIMD Extensions (SSE), first added 8 separate 128 wide registers

[1] In fact, memory access times depend on the internal clockings of the memory modules integrated in the computer as well as the clock frequency of the *Front Side Bus* (FSB), which connects it to the CPU. In "Principles of Computer Architecture" [MH99, p. 256] the access time is specified as 60-80 nanoseconds, whereas a register access costs only 1 nanosecond. Although the book is from the late 1990s and the data is likely to be outdated, the proportion should be still valid today.

for vector calculations (`xmm0` to `xmm7`), later complemented by another 8 (`xmm8` to `xmm15`) by SSE4. Besides, SSE introduced the possibility to process 4 single or 2 double precision floating point values in parallel, on which I will concentrate in the following. Most recently, Intel and AMD have begun to produce new processor series that include support for the so-called Advanced Vector Extensions (AVX), which can be considered the legal successor of SSE4. The main advantage of AVX over SSE4 is the introduction of 256 bit wide registers and corresponding instructions. Nevertheless, as AVX has only been available on the market for a few months and has not changed much with regards to vectorized floating point C code, I will only focus on SSE. The reader may replace "AVX" whenever I write "SSE".
SSE instructions on so-called *packed values* are executed on multiple, physically existing execution units and hence usually need the same amount of clock cycles as their scalar counterparts. For example, on a "Wolfdale" Intel Core 2 processor, both the `ADDPS` SSE instruction and the `FADD` instruction have a throughput of 1 instruction per clock cycle [Fog11a, pp. 50, 57]. Whereas `FADD` calculates the sum of 2 floating point values on the x87 floating point unit (FPU), `ADDPS` calculates the sums of 4 pairs of floating point values in two `xmm` registers. Listing 3 shows simplified assembler code that calculates the sum of an array of floats using SSE instructions. Note that this is AT&T syntax, i.e. source operand before destination operand.

Listing 3: Array sum using simplified SSE assembly

```
1   ; ecx contains the length of the array
2   ; edx contains the address of the array
3
4   movaps [edx], xmm0
5 LOOP1:
6   add    0x10, edx
7   movaps [edx], xmm1
8   addps  xmm1, xmm0
9
10  dec    ecx
11  jnz    LOOP1
12
13  haddps xmm0, xmm0
14  haddps xmm0, xmm0
15  movss  xmm0, ebx
16
17  ; now ebx holds the sum of the floats
```

The `movaps` instruction (`mov`e `a`ligned `p`acked `s`ingle) moves 16 bytes (or 4 single precision floats) between memory and a `xmm` register. `addps` adds 2 registers filled with packed single precision floats vertically. The most interesting part here is the SSE instruction found in lines 13 and 14: `haddps` horizontally adds adjacent elements in the two operand registers and stores the sums in the destination register, as can be seen in Figure 2. This illustrates that SSE code usually turns out to be considerably larger in size and number of instructions than scalar code, as the wrapping of data values into vectors (i.e., the "packing") and the un-wrapping again are distinct steps that are only present in vectorized code. In this case, the horizontal add would not be needed if the floats were accumulated one by one.
The theoretical speed-up factor when enriching scalar code with single precision floating point SSE instructions is limited by the maximum number of floats to be processed simultaneously, which is 4. The mentioned vectorization steps constitute a large enough overhead to easily reduce the speed-up to a factor of 3. However, this speed-up only applies to linear code without control flow statements depending on single data values. Since conditional branches (`if` statements and the like) inhibit data-parallel execution, they present a difficult to manage obstacle for

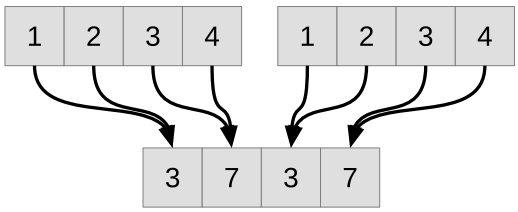


Figure 2: Horizontal Add Packed Single

SSE optimization, which needs to be cleared by using special techniques such as blending (see section 3.3.2). These may sometimes impose a performance drawback that completely consumes the expected vectorization speed ups. In practise, my experiments with SSE and moderately complex code resulted in actual speed up factors between 1.5 and 3 (see Section 4.6).

3.3 Techniques for programming with SSE

Fortunately, programming with SSE is well supported by modern compilers. Nowadays, there is no need to use inline assembler at all, as Intel, since the very beginning of MMX and SSE, provided specifications for *compiler intrinsics* for C/C++ languages along the SSE specifications. These intrinsics can be used to instruct the compiler to emit a specific SSE instruction, although the compiler is not obliged to actually comply with the programmer's request. For a start, there were two data types defined that help mixing SSE with legacy code: `__m128` and `__m128i`. The former maps to a 128 bit wide floating point vector, the latter a 128 bit wide integer vector. These can be seen as additional primitive data types that behave in the fashion of small fixed length arrays of the corresponding type. Recent versions (>= 4.6) of the GNU compiler collection even let the programmer access single elements using the well-known `[]`-operator. SSE intrinsic names follow consistent patterns: They all start with a `_mm_` prefix, usually followed by the name of the instruction, followed by the type of the vector's contents. For example, `_mm_hadd_ps` emits a "horizontal add packed single" instruction. It takes two `__m128` arguments and returns another `__m128` result value. Listing 4 demonstrates the use of SSE instrinsics for the array sum example given above.

Listing 4: Array sum using SSE instrinsics

```
1   __m128 sum = _mm_load_ps(&array[0]);
2   for(int i = 4; i < count; i += 4) {
3     __m128 add = _mm_load_ps(&array[i]);
4     sum = _mm_add_ps(sum, add);
5   }
6
7   sum = _mm_hadd_ps(sum, sum);
8   sum = _mm_hadd_ps(sum, sum);
9
10  // now sum[0] holds the sum of the floats
```

The GNU compiler collection recently added support for the usual arithmetic operators for vector types, so instead of writing `sum = _mm_add_ps(sum, add))`, we could also write `sum += add`. This especially helps to minimize readability loss incurred by SSE intrinsics. The use of intrinsics has several advantages over inline assembler. First, as mentioned before, the vector types can be treated as if they were primitives. For instance, the programmer need not care

whether they are stored in registers or in memory. The compiler inserts the appropriate `mov` instructions when they are needed for calculations. This even includes the possibility to create arrays of vector types. Besides, although almost all SSE instructions use a destructive destination operand (i.e., the second operand of an add operation is also used to store the result), intrinsics will let the compiler create copies of the required registers and thus let the programmer reuse his values. Second and even more important, the use of intrinsics leaves some space for compiler optimization. Whether it be register management or the order of instructions, the compiler may always find ways to further optimize the SSE code. In fact, the compiler may choose not to emit a requested instruction at all, if it sees fit. In a blog entry from 2009 [Lir09], the author "LiraNuna" compares the optimized assemblies of the three major SSE-aware compilers. Apart from revealing considerable differences between the produced assemblies, it also displays that all of the popular compilers were able to further optimize simple SSE intrinsics. For example, gcc succeeded in optimizing out 5 useless vector-based arithmetic operations (e.g., multiplying by a vector whose elements are all 1). In sum, there are rarely any situations where inline assembler has any advantages over the use of SSE intrinsics.
The preparations needed for the code to be of actual practical use are not shown in Listing 4. The array sum would not be correct if count was not a multiple of 4, as then the remaining 1, 2, or 3 elements would not take part in the calculation. There are two options suitable for fixing this issue: Either the programmer could add the remaining elements in a scalar loop right after the vectorized loop (with the iterator starting at the last multiple of 4 less than or equal to count), or he could always pad the array to a length dividable by 4. The padding elements would need to be initialized to zero. Either way would impose a substantial processing overhead (mainly caused by the additional loop) that would render the optimization useless for small arrays. Some other peculiarities of SSE programming are described in the following.

3.3.1 Aligned data storage

Most SSE instructions that operate on either registers or memory locations require those memory locations to be aligned by 16 byte boundaries (in other words, the memory address needs to be dividable by 16). For instance, `haddps` allows a memory location as source operand, yet will raise a so-called general protection exception when this location is not aligned by 16 bytes, resulting in an application error. For some SSE instructions such as data move instructions (`movaps`) there exist instruction variants that allow for misaligned addresses (e.g. `movups`). Shahbahrami et al. discussed the performance impact of misaligned accesses in [SJV06]. The authors point out that, as the processor must rotate or merge registers to support misaligned memory accesses, these accesses will always result in perceptible delays. Besides, misaligned accesses can lead to cache line splits as misaligned memory can be spread over multiple cache lines, which may produce additional slow downs[1]. Their experiments with the addition of two arrays with varying data types and lengths show that aligned accesses using SIMD instructions are on the average about twice as fast as misaligned accesses. The paper also demonstrates various techniques to avoid misaligned accesses.
When the programmer can control the allocation of memory himself and the memory should be allocated on the stack, he can easily tell the compiler to align the memory by 16 byte. In gcc this can be done by inserting `__attribute__ ((aligned(16)))` right after a variable declaration, both for primitives and arrays. The built-in SIMD data types are aligned automatically. Compiler guaranteed alignment of simple data types has the advantage of being totally costless at runtime

[1] In theory, the performance impact of a cache line split should not be any worse than that of an unaligned access. However, an interesting blog entry [GG08] of x264 developer Jason Garrett-Glaser suggests that cache line splits result in costly penalties on some Intel architectures (e.g., Core 2). Glaser carries on with explaining several SSE-related techniques to circumvent these penalties.

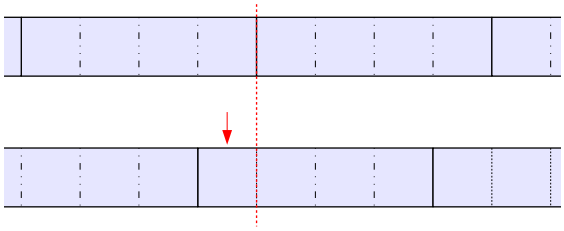


Figure 3: Example of cache line alignment and cache line split

apart from a possibly increased memory usage. When it is necessary to allocate memory on the heap, the programmer can use special aligning allocators such as `memalign` or `valloc` in POSIX compliant operating systems.
When the programmer has to deal with foreign memory, for example a return buffer from a library call, avoiding misaligned memory accesses will always come with a small processing overhead. When it comes to iterativly processing such a misaligned array, the probably most practical approach is called *loop peeling* or *loop splitting*. Here the relevant loop is split up into three parts: the main loop, in which the bulk of the array is processed using SSE instructions, is framed by two loops containing scalar code to process all elements before the first and after the last 16 byte boundary in the array. See Listing 5 for an example. Yet, these additional loops (especially their conditional jumps) amount to increased processing time that can effectively consume the expected performance gains in some situations. Another simple approach useful for situations with small data that gets created once but is read often would be to copy the data to an aligned memory address.

Listing 5: Loop peeling example

```
1   // Iterate to the first memory address dividable by 16.
2   float sum = 0;
3   int i = 0;
4   for(; (i < count) && (&array[i] % 16 != 0); i++)
5     sum += array[i];
6
7   // Vectorized main loop.
8   __m128 tmp = _mm_load_ps(&array[2]);
9   for(; i < count - (4 - i); i += 4) {
10    tmp += _mm_load_ps(&array[i]);
11  }
12  tmp = _mm_hadd_ps(tmp, tmp);
13  tmp = _mm_hadd_ps(tmp, tmp);
14  sum += tmp;
15
16  // Add remaining elements.
17  for(; i < count; i++)
18    sum += array[i];
```

3.3.2 Dealing with conditional branches

The by far hardest hurdle in SSE programming is the transcription of conditional branches. Conditional jumps depending on single values obviously can not exist in data-parallel code and thus need to be replaced by arithmetic or logical linear statements. For example, consider that

we wanted to sum up only those elements of the float vector (refer to Listing 4) that are greater than 5. In a scalar version this would require an `if` statement within the `for` loop that, when evaluated to `true`, would allow the current float variable to be added to the sum. Getting rid of the conditional jump in scalar C code is possible through evaluating the logical value of a comparison as an integer. Knowing that `false` is defined to be an integral zero and that `true` is equal to 1 in most C implementations, we could replace the `if` statement and therefore the conditional jump with: `sum += array[i] * (array[i] > 5)`. Comparisons in SSE also define zero as the value of `false`, but return `-NaN` as `true`, which translates to negative "Not A Number" or `0xFFFFFFFF`, a bitmask that has a 1 at all positions. Thus, whereas scalar C code requires a multiplication to eliminate the conditional jump, SSE takes a bitwise conjunction. See Listing 6 for a SSE implementation that calculates the sum of all array elements which are greater than 5.

Listing 6: Sum of array elements greater than 5

```
1  __m128 sum = _mm_setzero_ps();
2  __m128 five = _mm_set1_ps(5.0f); // (5.0, 5.0, 5.0, 5.0)
3  for(int i = 0; i < count; i += 4) {
4    __m128 add = _mm_load_ps(&array[i]); // e.g. (1.0, 6.0, 2.0, 8.0)
5    __m128 mask = _mm_cmpgt_ps(add, five); // -> (0.0, -NaN, 0.0, -NaN)
6    sum += _mm_and_ps(add, mask); // -> (0.0, 6.0, 0.0, 8.0)
7  }
```

If the conditional had an `else` branch containing a different assignment statement, the SSE code would require two additional bitwise logical operations, namely an `ANDNOT` for the `else` assignment and an `OR` to blend the results of the branches. With SSE4, this common triple of bitwise operations has been compacted into a single instruction called `blendv` (along with other types of blending operations) that merges the bits of two vectors based on a third mask vector.
In general, performances of vectorized code sections that work around conditionals largely depend on two factors: Probabilities of truth values and the amount of computation done in the related branches. For simple branches such as the assignment statement above it is usually true that a vectorized solution will outperform the scalar version as conditional jumps have a significant impact on performance. Whenever a conditional jump appears in application code, the processor tries to guess what path will be taken and precalculates that branch's results. When the guess turns out to be wrong (*branch misprediction*), the precalculation needs to be discarded and the processor's pipeline needs to be flushed, which can easily stall the processor for a hundred clock cycles. In contrast, vectorized code always evaluates both possible branches and blends the result according to a conditional bitmask, as explained above. As there are no conditional jumps, branch mispredictions become irrelevant. Yet, modern processors feature advanced branch prediction technology and often branch mispredictions are fairly uncommon. In some cases, this may even get to the point where a well predictable condition that determines whether a larger amount of code will be executed or not turns out to be faster than the corresponding vectorized code without condition that will always execute that code. As an extreme example, consider a basic `if` clause that evaluates to `true` only once in a thousand runs but then will lead to the execution of a considerable amount of code. Here, scalar code may surpass the performance of its vectorized counterpart. This issue also takes effect in long loops that contain "early out" conditionals (e.g. an early `if-break` group), since the loop needs to be executed until that conditional is `true` for all vector elements, when it may have canceled the loop for single elements much earlier.
When dealing with long code sections guarded by a conditional, the straightforward approach is to blacklist vector elements that should not take part in the calculation in a blacklist vector. Statements that have no outside effects can then be executed for the whole vector, while code sections that have outside effects need to be blended depending on the blacklist or need to be

performed during or after the unwrapping of the vectors. It may also be possible to omit the blacklist vector and instead directly incorporate the `NaN` value resulting from the comparison into the calculation. Any operation including a `NaN` value will always return another `NaN` value, so the initial `false` from the comparison will show up at the end of the calculation. However, this may only lead to a small speed-up in comparison to an explicit blacklist vector, mostly due to the blacklist likely having been dropped from cache. Occasionally it may even be feasible to omit the comparison, if the succeeding instructions will always lead to zero or infinity values for those vector elements that originally would have been sorted out by the comparison. However, as this is an arithmetic optimization, it might as well have been done in the original scalar code in the first place.
In case a conditional frequently evaluates to `false` for all vector elements, the new SSE4 `ptest` instruction comes to help. This instruction is especially valuable for the above mentioned case of rarely executed code sections after a conditional that is unlikely to be `true` at all. `ptest`, the first SSE instruction to operate on the 128 bit value as a whole, does a bitwise comparison of two `xmm` registers and returns a scalar `true` or `false`. Additionally, Intel provided some helpful convenience intrinsics that test a vector for "all 1" or "all 0" values which are named `_mm_test_all_zeros` and `_mm_test_all_ones`. See Listing 7 for two examples of `ptest` usage.

Listing 7: Examples of `ptest` usage

```
1  // Rarely true condition.
2  __m128 cmp = _mm_cmpgt_ps(a, b);
3  if(! _mm_test_all_zeros(cmp)) {
4    // Rarely executed code.
5  }
6
7  // Test whole vectors for equality.
8  // Cast intrinsincs do not actually emit an instruction,
9  // but are required for static type compatibility.
10 int equal = _mm_testc_si128(_mm_castps_si128(a), _mm_castps_si128(b));
```

3.3.3 Prefetching & non-temporal streaming

When trying to optimize the memory throughput of an application, the programmer can manually control cache management with prefetching and streaming. Prefetching, on the one hand, instructs the processor to preload memory into the caches that will be needed soon. This, however, rarely has any effect on performance as modern processors integrate sophisticated hardware prefetching technology. Yet, in some cases it may yield some improvements when used in the right way. Manual prefetch instructions should always point far ahead in memory (e.g. 500 bytes) and should only be issued when the programmer is absolutely confident that he will need the memory soon and that it has not been cached already. He also needs to make sure that prefetching does not displace already cached memory segments that are going to be used prior to the prefetched memory, which would result in a performance decline. This is commonly refered to as "cache thrashing" and may be considered a rare worst case scenario. Note, that the prefetch instruction is only a hint to the processor which it may or may not obey, in fact, the instruction may not do anything at all, when the processor is already busy loading different segments of memory. Prefetching will always require a fair bit of trial and error and in my experiments resulted in virtually zero performance improvements.
Streaming, on the other hand, is a rather common technique that can significantly increase memory throughput. The `movnt-` family of SSE instructions (e.g. `movntps`, meaning "move non-temporal packed single") hint to the processor that the data that should be moved to memory

will not be needed again soon. "Non-temporal" relates to the temporal characteristic of the before mentioned *locality of reference* (again, refer to Section 3.1.2) which is the foundation of today's cache architecture. When the processor encounters a `movntps` instruction, it will try to write the data directly to memory and bypass its caches. This has two major advantages over the regular move instructions: First, it is not necessary to read the memory segment to cache first, which results in noticeable speed-ups if the memory segment has not been loaded to cache already. Second, it will avoid polluting a cache line with pointless data, so that data that may be used again can remain in cache. Though, non-temporal data moves will only avoid caching when used with data big enough to fill up an entire cache line, i.e. at least 64 bytes on most systems. This is due to the fact the processor can only bypass the cache if it is able to use its "write-combining" buffers. Apart from that, a common advice is to issue a `mfence` instruction following the non-temporal streaming instructions, which stalls the processor until any memory operation has completed, in order to ensure that succeeding reads from that memory return the correct data. AMD has provided a helpful summary of how to properly use streaming instructions in their optimization manual [AMD12, pp. 106ff, 231ff]. Since SSE has become an industry standard that both Intel and AMD are committed to fully support, most of the information presented in this manual should apply to Intel processors as well.
In order to experience the speed-up of streaming store instructions myself, I wrote three different `memset` implementations for floating point values. Table 1 displays results of these experiments using different compiler optimization options (from `-O0` to `-O3`). As can be seen easily, the SSE implementation based on simple store instructions turns out to have no impact on performance at all with compiler optimization turned on, which is likely due to the compiler having auto-vectorized the loop in the scalar version. Streaming store instructions, by constrast, outperform the other implementations by factor 2. It seems impossible for the compiler to determine if the data written to memory is going to be reused soon, thus it does not emit streaming store instructions itself. The code of this benchmark can be found in Appendix A.

Table 1: Comparison of `memset` implementations

Implementation	Runtime (s), -O0	-O1	-O2	-O3
scalar memset	0.179	0.999	0.993	0.0987
SSE memset (`mov` instructions)	0.100	0.102	0.103	0.100
SSE memset (`movnt` instructions)	0.069	0.041	0.039	0.039

3.4 A word on compiler optimization

Whenever a programmer chooses to optimize a particular piece of code, he has to be aware of the fact that the compiler — when used with the right set of command line arguments — usually may have already created optimal machine code from that code. As a consequence manual code optimization very often will not yield the desired performance improvements but instead may even slow down the application. Besides, manual optimization will always be a time-consuming task and in the end lead to less readable code. Felix von Leitner gave a striking speech [Lei09] about compiler optimization at "Linux Kongress" conference 2009 in which he emphasizes that it is far more important to learn how compiler optimization works and how to structure code in a way that supports compiler optimization than to try to manually outperform the compiler. He concludes that only when the performance boost "drastically" outweighs the decrease in readability, manual optimization really is worth the effort. Again, I would like to

point out that the value of manual optimization depends on the very situation. Although on a small scale the compiler will most of the time create fast enough code and manual optimization may not improve performance at all, a programmer may find optimization measures at a larger scale that speed up execution a lot. The same is true for vectorization: Most modern compilers are able to auto-vectorize specific parts of code such as smaller loops without data dependencies. However, compilers are not able to grasp the complexity of longer loops that could be vectorized as well, for example by using branch avoidance techniques as described above.

4 Optimizing the LS^2 simulation engine

In the following section I will describe how I optimized the LS^2 simulation engine using mainly SSE intrinsics. I will begin introducing the software itself and its research purpose. Thereafter, I will document the approach taken for development, including information about how I benchmarked the application in order to acquire meaningful results. The main part of this section will comprise an overview of the optimized source codes of the three algorithms that I have chosen to improve. Code sections of particular interest with regards to special SSE programming techniques will be singled out and explained in detail.

4.1 LS^2: A simulation engine for lateration algorithms

The term "lateration algorithms" is commonly used to refer to geometric algorithms that use distance measurements to determine the location of points in the plane or in a three-dimensional space (as opposed to triangulation which uses the measurement of angles). The most basic representative of this group of algorithms is known as *Trilateration*: In the euclidean plane it needs at least three known spots (subsequently called *anchors*) and the distance measurements hereof to be able to narrow down the current position to a single point. Relative to each of the anchors, this point lies on a circle that has its center at the anchor position and the distance as its radius. Trilateration determines the current position by solving these three linear equations, in other words, it calculates the intersection of the three circles drawn around the anchors. In real-world applications such as the Global Positioning System (GPS) or indoor localization, distance measurements, like all physically measured data, are generally error-prone. Most commonly, distances are estimated by measuring the time it takes a signal (e.g. light, radio) to travel between an anchor and the client. Because of these erroneous distances, circles drawn around the anchors do not necessarily intersect at a single point and the basic trilateration algorithm fails to produce an exact result. In order to calculate an approximation of the current position, trilateration can be adapted to return the geometric center of the now up to six circle intersections. However, during the last decades, several superior, more complex algorithms have been found that compute improved position estimations based on error-prone distances of three or more anchors.
The *FU Berlin Parallel Lateration-Algorithm Simulation and Visualization Engine* (LS^2), written by Heiko Will, Thomas Hillebrandt, and Marcel Kyas, and first presented at the WPNC conference 2012, is a graphical evaluation framework for lateration algorithms. As the authors explain in [WHK12], the application's fundamental idea is to not only calculate an average algorithm error based on randomized locations, which has been the main evaluation criterium for lateration algorithms so far, but to calculate errors for all locations on a given "playing field" in parallel and in the end to provide the user with an image displaying the so-called *spatial position error distribution*. The assumption is that the position of the anchors has significant influence on the algorithm's performance, even more so than the errors of the distances. Figure 4 shows an exemplary output image created with the LS^2 engine using the VBLE-OPT algorithm and a

set of 3 anchors, only the anchor positions have been slighty magnified for better visibility. The left part of the image displays the average position error for every location (where bright colors indicate low average error), whereas the image part to the right displays the peak error for every position.

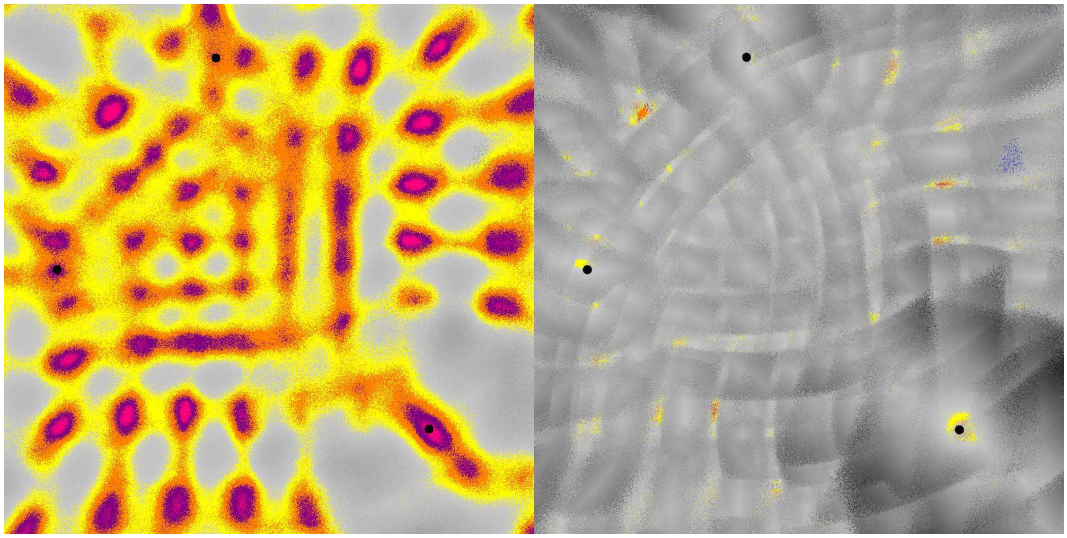


Figure 4: Example output of the LS^2 engine

The application is divided into three main parts: The engine itself, responsible for distributing the work load and calculating the position errors, a set of error models, which are used to simulate the distance measurement errors, and the lateration algorithms. Parametrized with a set of anchor positions, the error model, and the desired algorithm, the engine first starts a number of threads and associates them with a spatial slice of the 1000x1000 playing field. For each position in its slice, a thread calculates the real distance between the position and the anchors, randomly modifies the distances according to the error model, and executes the lateration algorithm. It then calculates the algorithm error as the difference between the real distance and the algorithm's return value. This process is repeated for a configurable number of iterations (defaults to 1000) for each position, before the resulting average error is written back to an image buffer. At the time of writing, LS^2 features 6 different lateration algorithms, of which 4 are standard algorithms such as trilateration or *Adapted Multilateration* (AML) and 2 are novel algorithms provided by the authors themselves. Regarding the error model, the user can choose between a uniform distribution and a Gaussian distribution of error values, and work is underway to support map-based error models as well.
The LS^2 engine is implemented in C99 dialect and is thoroughly optimized for speed. All functions are forcibly inlined and reside in the same compilation unit. Apart from the image buffer, no dynamically allocated memory is used. The engine itself uses SSE instructions or, at the user's wish, the newer AVX instructions available on Intel's most recent Sandy Bridge microprocessors, to process either 4 or 8 iterations at once. For example, the calculation of the true distances is fully vectorized, as is the random number generator used by the error models. Thereafter, the engine will execute the user-requested lateration algorithm with vectorized parameters. To understand the mechanics, refer to Listing 8 which shows the function definition of the trilateration algorithm. The `count` parameter specifies the number of anchors the algorithm should use, whose x- and

y-coordinates are contained in the `vx` and `vy` arrays. Note that these, as well as the distance array `r` and the result buffers `resx` and `resy`, are declared as `VECTOR`s, which means that there are 4 (resp. 8) of each of these values waiting to be processed at once. Note that `VECTOR` is a preprocessor `define` mapping to `__mm128` when the application is compiled in SSE mode and to `__mm256` in AVX mode.

Listing 8: Prototype of the `trilaterate` function

```
1 void trilaterate(const int count, const VECTOR* vx,
2                   const VECTOR* vy, const VECTOR* r,
3                   VECTOR* resx, VECTOR* resy);
```

When I started looking into optimizing the LS^2 application for this thesis, some of the included algorithms, for example trilateration and the *Linear Least Squares* algorithm, already made use of SSE instructions to process their parameters in a single run. Yet, some others were merely literal transcriptions from a scalar implementation and were not aware of vector processing at all. These algorithms (*Geolateration*, *Optimized Voting Based Location Estimation*, and *Adapted Multilateration*) simply unwrapped the vectors at the function head, calculated the position estimations in scalar code and packed the results back into the result buffers at the end of the function. Therefore, these three algorithms best qualified for having a deeper look into their optimization potential.

4.2 Development approach

During the period of development, it quickly became clear to me that software optimization is not an utterly straightforward task, but a rather non-linear process instead, involving mostly trial and error methods. Quite often an attempt to to optimize a particular piece of code using a specific optimization technique would not lead to any performance gains, even though in theory it may have looked like a very promising measure. Yet sometimes, these attempts that seemed to be "dead-ends" at first try later became valuable complements to other optimization efforts. As a consequence, my "development process", which was anything but well-defined at the beginning, turned out to be slightly different from regular iterative processes. As it is mainly a report on my personal experiences, the following section should not be misinterpreted as a universal guideline. The described steps and tools simply suited my needs best.

4.2.1 Determining bottlenecks through profiling

As Donald E. Knuth has already stated memorably in 1974, it is a good idea to first determine the most performance-critical part of a piece of code before beginning with optimizational work, as this is likely to be the only part where optimization really is worth the effort. To find that critical part, the performance bottleneck, profiling a software can be helpful. For C/C++ code and on Unix-like platforms, *Valgrind*[1] has become the de-facto standard profiler utility. The Valgrind suite consists of a variety of specialized profiling tools, each of them named differently, as for instance a memory usage analyzer (Memcheck) that especially helps finding memory leaks, a heap profiler (Massif) for dynamic memory profiling, and a cache profiler (Cachegrind) that simulates cache utilizitation to detect cache misses. The latter is especially valuable for optimization, as it additionally counts the executed instructions for each code line and is able to estimate the amount of clock cycles the processor needs to execute them. On top of that, Cachegrind features a special option (`-branch-sim=yes`) that, when enabled, lets Valgrind count branch mispredictions and

[1] `http://valgrind.org`, last accessed: May 4, 2014

thus can be used to identify particularly unpredictable branches. Cachegrind's result can be best viewed using the *KCachegrind*[1] front-end, as demonstrated by the screenshot in Figure 5.

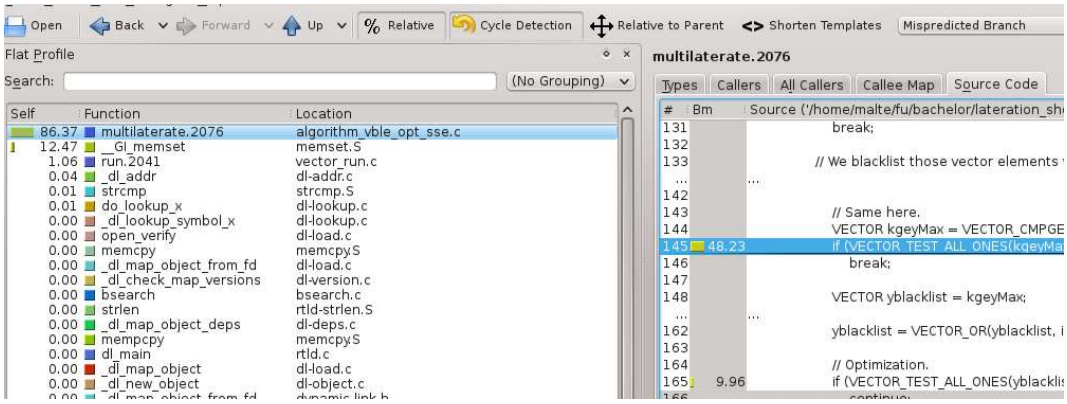


Figure 5: KCachegrind, branch prediction view

4.2.2 Benchmarking optimization efforts

After having identified the bottlenecks and resolved them with the help of optimization techniques, the next step is to evaluate their impact on performance. Before and after every smaller step, I used the unix `time` command to measure performance of the application for a randomly selected set of input values. As can be read on the command's manpage, `time` measures three different execution times (see Listing 9 for an example output). The "user" value indicates the amount of CPU time spent in user-mode, the "sys" value is the amount of CPU time spent in kernel-mode. The "real" value is the real time that has elapsed between start and termination of the application. For threaded applications such as LS^2, on dual-core systems this should ideally be about half of the combined "user" and "sys" times plus some overhead for context switches. Since the "real" time varies with the number of processors available and the number of system-calls contained in LS^2 is close to zero, the only information that made a difference for my optimizations was the "user" value.

Listing 9: Example output of the unix `time` command

```
1 $ time { ./bin/lateration_shooter_sse 100 400 500 200 700 800 ; }
2 Average error is 45.252751
3
4 real    0m8.056s
5 user    0m14.916s
6 sys     0m0.008s
7 $
```

As the performance of the various algorithms in LS^2 turned out to be highly affected by the positions of the anchors, it was also crucial to evaluate the performance gains using a larger amount of randomized input data in order to avoid optimizing the application for a single case. For this time-consuming task I wrote an increasingly useful Python script called *benchlat*, which automatically compiled my various work states, benchmarked them a larger number of times

[1] `http://kcachegrind.sourceforge.net/html/Home.html`, last accessed: May 4, 2014

overnight, and sent an email after completion containing the analyzed results, namely average and variance of the achieved speed-ups. Individual speed-up ratios were obtained by running the reference implementation and my optimized versions with the same set of anchor positions. Later I added automatic profiling using Cachegrind as well as an option that allowed tracing the performance fluctuation through the development history.

4.2.3 Managing various attempts using a version control system

As mentioned before, optimizing LS^2 was an unsteady task that was characterized by hours of trial and error. When an optimization proved to be the best solution in the very situation and to measurably improve performance, I integrated it into the sources and moved on to the next chokepoint. These successes were often preceded by a number of less profitable attempts that I wanted to save for possible later reuse. Making heavy use of lightweight branches as provided by a modern version control system such as Git[1] proved highly beneficial for these cases. Using branches, I could temporarily put unpromising changes aside and merge them back in when they again seemed practical solutions in combination with other work. To illustrate the trial and error process more vividly, Figure 6 displays a schematic overview of the development in form of a visualization of the complete Git history. Nodes represent the numerous Git commits, which are linked by arrows indicating an "ancestor-of" relationship. As can be seen easily, the development path was not very straightforward.

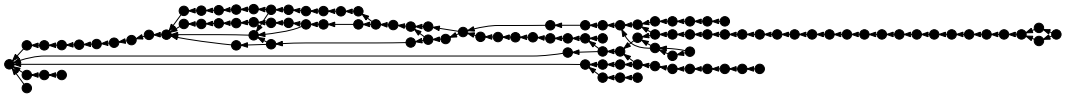


Figure 6: Visualization of `git` history

4.3 Algorithm I: Adapted Multilateration

Proposed by Kuruoglu et al. in [KEO09], Adapted Multilateration (AML) is a lateration algorithm that produces acceptable results while being of lower complexity. As in the case of trilateration, it is based on anchor circles and their intersections.

4.3.1 Functionality

AML consists of three steps: *intersection and elimination*, *first estimation*, and *refinement*. At the beginning of step 1, two intersecting anchor circles are chosen randomly. If these circles intersect at exactly one point, that point is picked for step 2. In case of two circle intersections, one of them needs to be eliminated, which is the one that has the larger distance to the circle of a third anchor. Afterwards, a first estimation of the position is obtained by moving the intersection point to the middle of the line segment connecting it with the third anchor's circle (i.e. with the closest point on that circle). Any remaining anchors are processed in the same way in the final refinement step: For each anchor, the current location estimation is moved to the middle of the shortest line segment connecting it with the anchor's circle. Refer to the paper for a more extensive and more graphical explanation of the algorithm.
To calculate the said movement, the authors state it would be necessary to differentiate whether the current point is located on the inside or on the outside of the anchor's circle and treat these

[1] `http://git-scm.com/`, last accessed: May 4, 2014

cases differently. However, I could not spot any reason why this method would be advantageous over using simple vector mathematics[1]. Still, as algorithmic optimization was not part of my goals, the optimized code presented in the following corresponds to the author's original algorithm description.

4.3.2 Optimizations

Since the AML implementation is concise and easily understood, I will be able to present it in its entirety in this section. For a more generic start, Listing 10 displays the already vectorized `circle_get_intersections_v` function, which, as its name implies, is used to calculate the intersections of two random anchor circles in step 1 of AML, yet it is written to process 4 pairs of circles at once, or 8 in the case of AVX. For reference, I included the original scalar implementation of this function in Appendix B. The function is based on the usual geometric approach of constructing a line through the intersections and using triangles to determine the offset of the intersections from the line segment connecting the anchors (refer to [Bou97]) for further explanation). Most interesting in the SSE implementation is the treatment of special cases occurring when the circles do not (properly) intersect: First, the circles may be disjunct, i.e. the distance between the anchors is greater than the sum of the circles' radii, second, one circle may be contained within the other (the absolute difference of the radii is less than the distance between the anchors), and third, the anchors may be coincident (the distance is 0), in which case there are 0 or an infinite number of intersections. In the original implementation these special cases were handled by an `if` statement that contained three boolean expressions linked by logical OR, each of them corresponding to one of the cases. In case the `if` statement evaluated to `true`, the function would immediately return 0. To eliminate the conditional jump, the SSE version inverts the boolean expressions and blends the `retval` vector to 1 in case all expressions are true, 0 otherwise (ll. 11–13 in Listing 10). Afterwards, all the usual calculations are performed regardless of whether the circles contained in the particular vectors truly intersect, which may result in unusual values for vector elements that failed the blending before. For example, coincident anchors, having a distance of 0, will lead to a division by zero in line 15, which results in variable `a` containing the floating-point-specific value `+Inf`, or "positive infinity". In fact, all intermediary results should be considered "tainted" until they are verified by re-evaluating the mask that is now stored in `retval`. An example can be seen in line 33: If variable `h`, which contains the distance between the intersection(s) and the line segment connecting the anchors, is non-zero, the algorithm has found two intersections. In this case, `retval` is incremented to indicate two solutions, but only when it already holds a 1, or in other words, when all special cases have been ruled out. As a noteworthy side-effect to be kept in mind by the programmer, the SSE implementation always modifies the `resx` and `resy` vectors, whereas the scalar implementation did not touch them if the circles did not intersect.

[1] Let $\vec{p}$ be the vector belonging to the intersection point and $\vec{x}$ the vector pointing at the center of the circle (see figure 3 in [KEO09, p. 264]), while $\vec{xp} = \vec{p} - \vec{x}$ is the direction vector from Point p to the center point x. We could then determine the intersection a of the line segment between x and p by scaling this direction vector to the circle's radius which means multiplying the vector by $\frac{r}{|\vec{xp}|}$, i.e. as a vector: $\vec{a} = \vec{x} + \frac{r}{|\vec{xp}|} \cdot \vec{xp}$. The middle of the line segment $\overline{pa}$ can be calculated as $\vec{p'} = \vec{p} + \frac{1}{2} \cdot \vec{pa} = \vec{p} + \frac{1}{2} \cdot \left(\left(\vec{x} + \frac{r}{|\vec{xp}|} \cdot \vec{xp}\right) - \vec{p}\right)$.

Listing 10: Calculating circle intersections with SSE

```
1  /**
2   * Returns the intersection of two circles for all 4/8 circles in the vectors.
3   * @ret: the number of intersections [0,1,2] of the corresponding circles.
4   */
5  FUNCTION VECTOR
6  circle_get_intersection_v (VECTOR p1x, VECTOR p1y, VECTOR p2x, VECTOR p2y,
       VECTOR r1, VECTOR r2, VECTOR* retx, VECTOR* rety) {
7      VECTOR d = distance(p1x, p1y, p2x, p2y);
8
9      // Mark not intersecting circles
10     VECTOR retval = VECTOR_AND(one, VECTOR_CMPGE(d, VECTOR_ZERO()));
11     retval = VECTOR_AND(retval, VECTOR_CMPGE(r1 + r2, d));
12     retval = VECTOR_AND(retval, VECTOR_CMPGE(d, VECTOR_MAX(r1 - r2, r2 -
           r1)));
13
14     VECTOR a =  (r1*r1 - r2*r2 + d*d) / (two * d);
15     VECTOR v = r1*r1 - a*a;
16     VECTOR h = VECTOR_SQRT(v);
17
18     VECTOR dx = (p2x - p1x) / d;
19     VECTOR dy = (p2y - p1y) / d;
20
21     VECTOR p3x = p1x + a * dx;
22     VECTOR p3y = p1y + a * dy;
23
24     dx *= h;
25     dy *= h;
26
27     retx[0] = p3x + dy;
28     rety[0] = p3y - dx;
29     retx[1] = p3x - dy;
30     rety[1] = p3y + dx;
31
32     retval += VECTOR_AND(retval, VECTOR_CMPGE(h, VECTOR_ZERO()));
33     return retval;
34 }
```

In order to find two random intersecting anchor circles, the previous implementation of AML simply tried all possible anchor combinations beginning with the first two anchors. However, since it is not guaranteed that the SSE version finds suitable anchor pairs for all vector elements in the same loop iteration, it has to continue the search until this condition is satisfied for all vector elements. Listing 11 shows the vectorized implementation. In line 11 the `mask` vector is constructed basing on two conditions: First, the loop must not have found a suitable pair in a previous iteration (`icount == 0`), and second, the current iteration must have succeeded in finding one (`icount_tmp > 0`). Depending on the `mask` vector the intersections and the anchor indices are blended into the corresponding buffers afterwards (ll. 13–14). The conditional statement in line 16, which uses the new `ptest` instruction introduced by SSE4, presents a so-called "early-out" option which breaks the loop in case intersecting circles have been found for all vector elements. Although this is optional with regards to the loop's correctness, it constitutes a major speed-up for the algorithm. In my experiments the loop found suitable anchor pairs for all vector elements in its first iteration in the majority of cases. Still, since this is likely to change with upcoming error models, the performance impact of the early-out option needs to be reevaluated in the future.

Listing 11: *Intersection* step of AML, vectorized version

```
1  FUNCTION void
2  aml_run (const int count, const VECTOR* vx, const VECTOR* vy, const VECTOR*
       r, VECTOR* resx, VECTOR* resy) {
3      VECTOR isectx[2], isecty[2], ci, cj;
4      VECTOR icount = VECTOR_ZERO();
5
6      // find two circles (random), which intersect in one or two points
7      for (int i = 0; i < count - 1; i++) {
8          for (int j = i + 1; j < count; j++) {
9              VECTOR isectx_tmp[2], isecty_tmp[2];
10             VECTOR icount_tmp = circle_get_intersection_v(vx[i], vy[i],
                   vx[j], vy[j], r[i], r[j], isectx_tmp, isecty_tmp);
11             VECTOR mask = VECTOR_AND(VECTOR_CMPEQ(icount, VECTOR_ZERO()),
                   VECTOR_CMPGT(icount_tmp, VECTOR_ZERO()));
12
13             icount = VECTOR_BLENDV(icount, icount_tmp, mask);
14             // [Skipped: Blend ci, cj, isectx, and isecty depending on mask.]
15
16             if (VECTOR_TEST_ALL_ONES(VECTOR_CMPGT(icount, VECTOR_ZERO())))
17                 goto endloop;
18         }
19     }
20     endloop:
```

The algorithm is continued in Listing 12 with a scalar code section that only serves the simplicity of the succeeding steps and, by implication, their performances, too. It purges the arrays containing the anchors and distances by removing those that have already been used within the *intersection* step of AML. This is a vivid example for code that can neither easily nor profitably be transformed into a vectorized representation, because it is inherently not data-parallel. Note that, while the `k` variable is advanced steadily, the "write buffer" index `n` needs to be incremented only when an unused anchor has been stored in the buffers, which inhibits moving the anchors in a vector instead of individually. Since the inner loop is fairly short, the unwrapping of the vectors imposes a considerable overhead. In fact, Cachegrind reports this code section to be the most cost-intensive part of the whole vectorized implementation.

Listing 12: Preparing refinement anchors

```
1      // Refinement anchors.
2      int rcount = count - 2;
3      VECTOR rvx[rcount], rvy[rcount], rr[rcount];
4      for (int ii = 0; ii < VECTOR_OPS; ii++) {
5          // store unused anchors for refinement step
6          for (int k = 0, n = 0; k < count; k++) {
7              if (k != ci[ii] && k != cj[ii]) {
8                  rvx[n][ii] = vx[k][ii];
9                  rvy[n][ii] = vy[k][ii];
10                 rr[n][ii] = r[k][ii];
11                 n++;
12             }
13         }
14     }
```

The first step of AML is concluded by eliminating one of the two possible circle intersections. Since intersecting in two points is most probably the case for all vector elements, this is again implemented using vector operations and blending, as shown in Listing 13. Lines 3–5 offer a different method of computing the absolute value of 4 packed floats, this time using a logical

`andnot` operation with a vector that has only the sign bits of all floats set to 1. Compare to line 12 in Listing 10 for another method. Vector `emask` indicates whether the vectors produced two intersections and, if so, which one is closer to the circle of a third anchor circle. Depending on this mask the applicable intersection point is blended into the `p_isectx` and `p_isecty` vectors, which are used as the starting point for refinements in the succeeding steps. As these variables are later returned as the results of the algorithm, lines 11–13 make sure that they contain `NAN` (i.e., no result) in case the initial *intersection* step could not find two suitable anchor circles. All subsequent calculations will leave these `NAN` values unchanged.

Listing 13: *Elimination* step of AML

```
1     // Starting point for refinements.
2     // Do elimination step in case of two intersections.
3     VECTOR mzero = VECTOR_BROADCASTF(-0.0f);
4     VECTOR delta_p1 = VECTOR_ANDNOT(mzero, distance(isectx[0], isecty[0],
          rvx[0], rvy[0]) - rr[0]);
5     VECTOR delta_p2 = VECTOR_ANDNOT(mzero, distance(isectx[1], isecty[1],
          rvx[0], rvy[0]) - rr[0]);
6     VECTOR emask = VECTOR_AND(VECTOR_CMPGT(icount, one),
          VECTOR_CMPGT(delta_p1, delta_p2));
7     VECTOR p_isectx = VECTOR_BLENDV(isectx[0], isectx[1], emask);
8     VECTOR p_isecty = VECTOR_BLENDV(isecty[0], isecty[1], emask);
9
10    // If no intersection, return NAN.
11    VECTOR vnan = VECTOR_BROADCASTF(NAN);
12    p_isectx = VECTOR_BLENDV(isectx, vnan, VECTOR_CMPEQ(icount,
          VECTOR_ZERO()));
13    p_isecty = VECTOR_BLENDV(isecty, vnan, VECTOR_CMPEQ(icount,
          VECTOR_ZERO()));
```

The final steps of AML, *first estimation* and *refinement*, are identical with regards to their computation and therefore combined in a single loop in the implementation. Listing 14 displays the case differentiation implemented as described in the original paper [KEO09, p. 264], which could as well be avoided as pointed out earlier. However, as all SSE programming techniques incorporated here should be well-known by now, I will skip the details.

Listing 14: *First estimation* and *refinement* steps of AML

```
    // do first estimation step and refinement
    for (int k = 0; k < rcount; k++) {
        VECTOR dist = distance(p_isectx, p_isecty, rvx[k], rvy[k]);
        VECTOR delta_pi = dist - rr[k];

        // case 1: point outside of circle
        VECTOR drr1 = (delta_pi / two) / (delta_pi + rr[k]);
        VECTOR delta_x_out = (rvx[k] - p_isectx) * drr1;
        VECTOR delta_y_out = (rvy[k] - p_isecty) * drr1;

        // case 2: point inside of circle
        delta_pi = VECTOR_MAX(dist - rr[k], rr[k] - dist);
        VECTOR drr2 = (delta_pi / two) / rr[k];
        VECTOR div = one - two * drr2;
        VECTOR delta_x_in = ((p_intersectx - rvx[k]) * drr2) / div;
        VECTOR delta_y_in = ((p_isecty - rvy[k]) * drr2) / div;

        VECTOR mask = VECTOR_CMPGT(dist, rr[k]);
        VECTOR dx = VECTOR_BLENDV(delta_x_in, delta_x_out, mask);
        VECTOR dy = VECTOR_BLENDV(delta_y_in, delta_y_out, mask);

        p_isectx += dx;
        p_isecty += dy;
    }

    *resx = p_isectx;
    *resy = p_isecty;
}
```

4.4 Algorithm II: Geolateration (Geo3)

Being one of the novel algorithms created by the authors of LS^2 themselves, Geolateration has yet to be introduced to a wider audience. It is implemented in LS^2 as a precursor version called *Geo3*, which only employs three anchors to estimate the current position. The authors are still working on the final version that will be able to utilize an arbitrary number of anchors and is referred to as *Geo-N*. As proper documentation has not been published yet, the descriptions below are based solely on discussions with the authors and my observations from the code itself. Geolateration also uses intersections of anchor circles as its foundation, yet compared to AML it takes a more spatially oriented approach to derive the position estimation hereof. Whereas AML values all circle intersections equally and in general is heavily influenced by the choice of the initial anchor pair, Geolateration tries to select a best-fitting subset of the intersections by assessing their spatial distribution using triangle geometry.

4.4.1 Functionality

As mentioned above, Geo3 is only capable of utilizing three anchors. Given a larger number of anchors, this triple is chosen randomly, in fact, the implementation always uses the first three anchors in the array. The first step of Geo3 comprises calculating all possible circle intersections of these anchors. In case two circles do not intersect an intersection point is estimated as the middle point of the line segment connecting the two closest points on the circles, i.e. the point where the circles would first touch if they were inflated. After the intersection calculation and approximation the algorithm holds between 3 and 6 points of interest. In step 2, which is mainly

an optimization, it is tested whether there exists a subset of 3 points which are "very close together" (pairwise distances of these points must be less than 0.1 simulation units), in which case one of these points is immediately returned as the result. Step 3 constitutes the geometric assessment of the intersection points: Out of all combinations of 3 points it finds the triangle with the smallest perimeter and the (possibly non-existent) smallest perimeter triangle which lies in all circles. See Figure 7 for a graphical illustration of these triangles. Here, the green triangle to the upper right has the minimal perimeter of all triangles that can be constructed from the intersection points (marked by blue dots) and the red dashed triangle is the triangle with the smallest perimeter that is fully contained in all 3 circles.

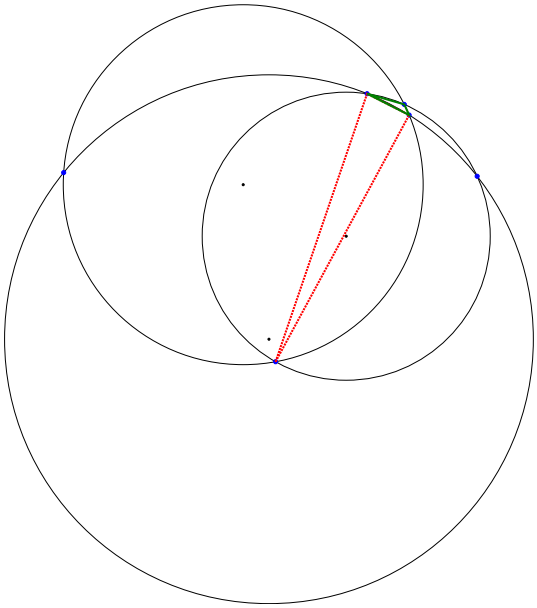


Figure 7: Minimum perimeter triangle and minimum triangle lying in all circles

Step 4 is again an optimization step. The algorithm detects whether the minimum perimeter triangle lies in the center of the triangle formed by the three anchors by comparing the distances of the triangles' centers of mass to the in-circle radius of the latter triangle. If it does, the center of mass of the minimum perimeter triangle is returned as the result. Afterwards, final steps 5 and 6 calculate the position estimation based on the following conditions:

- If the minimum perimeter triangle and the minimum perimeter triangle contained in all circles are the same or if the latter does not exist, return the geometric median of the minimum perimeter triangle's coordinates.
- If the triangles differ, evaluate their areas: If the latter triangle's area is "roughly equal" to the area of the minimum perimeter triangle, return the geometric median of that triangle's coordinates.
- If the areas are diverging more than a certain factor, calculate and return the geometric median of all 6 coordinates weighted by the area of their corresponding triangle.

4.4.2 Optimizations

Due to the many early-out conditionals in the various steps of Geo3, my attempts to vectorize the whole algorithm did not produce the desired performance gains. Instead, I decided to keep unwrapping the vectors provided by the engine at the beginning of the algorithm and only use SSE to enhance the following scalar code where possible. Apart from that, the calculation of the intersections themselves and the calculation of the incircle radius of the anchor triangle could be moved out of the loop in order to reduce redundant computation and additionally have been vectorized, using the known `circle_get_intersections_v` function for all circle intersections. In the following I will explain some remarkable parts of the optimized implementation of Geo3, starting with the precalculation of the distances between the intersection points.

Precalculating intersection distances. The distances between the circle intersections are needed for both step 2 and step 3 and have previously been calculated ad hoc when they were used, which resulted in a magnitude of wasteful computation. Therefore it seemed promising to precalculate and store these distance values, which can be as many as 36 including symmetric and reflexive intersection pairs. This was the first time I actually had to differentiate between SSE4 and AVX coding, as the larger registers of AVX allowed for a much simpler solution than SSE did. Within the 256 bit wide `ymm` registers available on AVX-aware processors I could easily store up to 6 intersection points and thus calculate their distance values to one of them in a single call to the vectorized `distance` function. However, since those 6 points do not fit into a 128 bit wide `xmm` register of SSE and the SSE version of the `distance` function also processes only 4 values at a time, I needed to use two registers and shuffle data around in order to reduce the number of additional calls to this function, which is rather expensive due to several multiplications and a square root operations. See Listing 15 for the code, as well as Appendix C for the AVX version. The `_mm_load_ps` instructions in line 6 load the leading 4 float values from the `ix` and `iy` arrays containing the intersection points into the `{x,y}tlower` vectors. The SSE `MOVLHPS` instruction moves the lower two words of the second parameter register to the upper positions of the first parameter registers (i.e., `MOVLHPS((a0,a1,a2,a3), (b0,b1,b2,b3)) == (a0,a1,b0,b1)`). The intrinsic for this instruction, `_mm_movelh_ps`, is used in lines 8–9 to fill the `{x,y}tupper` vectors with the fifth and sixth intersection point at both the lower and the upper two word positions. In other words, it simply copies the lower two words of both vectors to the corresponding upper positions. The loop starting in line 11 iterates the intersection points in steps of 2, first calculating the distances between those and the "lower" 4 intersection points. Then, the point vectors are merged in lines 20–21 to calculate the distances between the two points and the "upper" 2 intersections in a single call in line 22. This shuffling approach may seem overly complex, but in practise it achieved a performance improvement that fairly justifies the decrease in readability. This improvement mainly results from the reduced number of calls to the `distance` function. Note that the general assumption is that in the majority of cases the algorithm has to deal with the maximum of 6 intersection points, which may become less likely with different error models. The sole purpose of the `union` declared in lines 2–5 is to provide for transparent access to the distances after SSE and AVX code paths have been joined together.

Listing 15: Calculating distances for Geo3 (SSE)

```
1  // Precalculate distances
2  union {
3      VECTOR v[2];
4      float f[8];
5  } distances[6];
6  VECTOR xtlower = _mm_load_ps(ix), ytlower = _mm_load_ps(iy);
7  VECTOR xtupper = _mm_load_ps(&ix[4]), ytupper = _mm_load_ps(&iy[4]);
8  xtupper = _mm_movelh_ps(xtupper, xtupper);
9  ytupper = _mm_movelh_ps(ytupper, ytupper);
10
11 for(int i = 0; i < sum; i+=2) {
12     VECTOR px = VECTOR_BROADCAST(&ix[i]);
13     VECTOR py = VECTOR_BROADCAST(&iy[i]);
14     VECTOR px2 = VECTOR_BROADCAST(&ix[i + 1]);
15     VECTOR py2 = VECTOR_BROADCAST(&iy[i + 1]);
16
17     distances[i].v[0] = distance(px, py, xtlower, ytlower);
18     distances[i + 1].v[0] = distance(px2, py2, xtlower, ytlower);
19
20     px = _mm_movelh_ps(px, px2);
21     py = _mm_movelh_ps(py, py2);
22     distances[i].v[1] = distance(px, py, xtupper, ytupper);
23
24     distances[i + 1].f[4] = distances[i].f[6];
25     distances[i + 1].f[5] = distances[i].f[7];
26
27     // [Skipped: Set distances[i..i+1].f[sum..8] to FLT_MAX.]
28 }
```

Step 2: Looking for *close points*. Having precalculated the distances between the intersection points, it is now easier to determine whether there exist 3 points which are very close together. While the previous implementation needed a nested loop to count close points for each point, in the optimization version it is sufficient to horizontally add the distance vectors for each point and test whether the sum is greater than or equal to 3. Listing 16 shows the SSE implementation of step 2 of Geo3. The difference to the AVX version mainly consists of the extra comparison and addition operations in line 4, which are not needed for AVX as the distances reside in a single 256 bit vector, as well as a missing third horizontal add instruction. Both the fewer distance calculations and the elimination of the nested loop contributed largely to the overall performance improvement.

Listing 16: Looking for 3 close points (SSE)

```
1  VECTOR max_distance = VECTOR_BROADCASTF(0.1);
2  for (int i = 0; i < sum; i++) {
3      VECTOR tmp = VECTOR_AND(one, VECTOR_CMPLT(distances[i].v[0],
           max_distance));
4      tmp += VECTOR_AND(one, VECTOR_CMPLT(distances[i].v[1], max_distance));
5      tmp = _mm_hadd_ps(tmp, tmp);
6      tmp = _mm_hadd_ps(tmp, tmp);
7      if (tmp[0] >= 3)
8          // [Skipped: Return (ix[i], iy[i]) as result.]
9  }
```

Step 3: Minimum perimeter triangle. Step 3 also profitted greatly from the distance precalculation, as in order to get the triangle perimeters, the original implementation computed the distances repeatedly for each possible triangle. The search for the minimum perimeter triangle remains a major bottleneck in the optimized implementation though. Theoretically, the algorithmic approach has a asymptotic complexity of $O(n^3)$, since it has to process all possible 3-combinations of a set of points. Yet in practise, the implementation uses 3 nested loops with "staggered" initial values, i.e. the first loop starts at `i = 0`, the second loop at `j = i + 1` and so on. When applied to a maximum total of 6 points, this results in at most 120 iterations, as compared to the theoretical maximum of $6^3 = 216$. Trying to vectorize these loops and thus trying to access the intersection point arrays in a more sequential way would lead to the total number of tests growing closer to that maximum. Additionally, vectorization would require a lot of vector wrapping and unwrapping and data shuffling operations that will impose a significant performance overhead. In other words, algorithms which iterate over k-combinations of a set of n things will hardly benefit from vectorization when n is small.
In sum, vectorization was not an option for Geolateration's step 3. Some speed up could be achieved by vectorizing the `circle_point_in_circles` function which determines whether a point lies in all circles of a given set and additionally moving this calculation out of the loop. Still, this part remains a major time-consumer in Geo3.

Step 5: Computing the geometric median using Weiszfeld's algorithm. The final step of Geolateration involves calculating the geometric median of one or both of the two triangles' vertices. The geometric median (also known as Fermat-Weber point or 1-median) of a set of points is defined as the point for which the sum of the distances between this point and the points in the set becomes minimal. Since there exists no simple formula to calculate the geometric median, the authors implemented an iterative approximation approach known as Weiszfeld's algorithm [WP09]. I was able to improve the performance by deploying an SSE-enhanced version of this algorithm for up to 4 points[1], which is only used when the median of only one of the triangles needs to be calculated. For the statistically rare case when the median of two triangles should be calculated, it would be possible to translate the optimized implementation to AVX instructions, although I did not put this plan into practise. However, using SSE instructions for the two triangle case seemed inappropriate as it would require either twice as many instructions or a multitude of shuffling operations, similarly to the distance precalculation code described above. With regards to SSE programming techniques, the initial step of the algorithm is especially noteworthy, which is displayed in Listing 17. Put simply, this step tests whether one of given points is already "close enough" to the geometric median, in which case the remaining iterative approximation is skipped. The critical point is located in lines 6–7. Considering that the distance from point (`xi`,`yi`) to itself is always zero and that in addition some elements of the `ptsx` and `ptsy` vectors may also be zero as they store the user-defined parameters to the geometric median function, the elements of the `tmpx` and `tmpy` vectors may attain several special values resulting from the division operations: They may become positive infinity in case the distance value is zero and the `xi` or `yi` value is positive, negative infinity if the distance value is zero and the point value is negative, or `NaN` if both values are zero (i.e., in C, $0/0$ = `+NaN`). As these values must not be included in the following accumulation, they need to be detected and explicitly set to back to zero. Whereas in the original scalar implementation a conditional guarded the following code, in the SSE implementation this is accomplished using comparison instructions in

[1] It is actually possible to arithmetically construct the geometric median of a triangle, which is also referred to as the *Fermat point*. Since the optimized geometric median function is used exclusively for triangles in Geo3, it should probably rather have been replaced by another algorithm. I was not aware of this fact when I optimized Geo3.

lines 8–11, which assert that the `tmpx` and `tmpy` vector elements are valid floating point values between `+Inf` and `-Inf`. Note that any comparison applied to a `NaN` value always returns `false`. One may argue that the `tmpy` vector has not been checked for special values and thus may still corrupt the subsequent calculation, however, since all special cases result from a distance value (i.e., the divisor) being zero, either none or both of `tmpx` and `tmpy` may attain these special values.

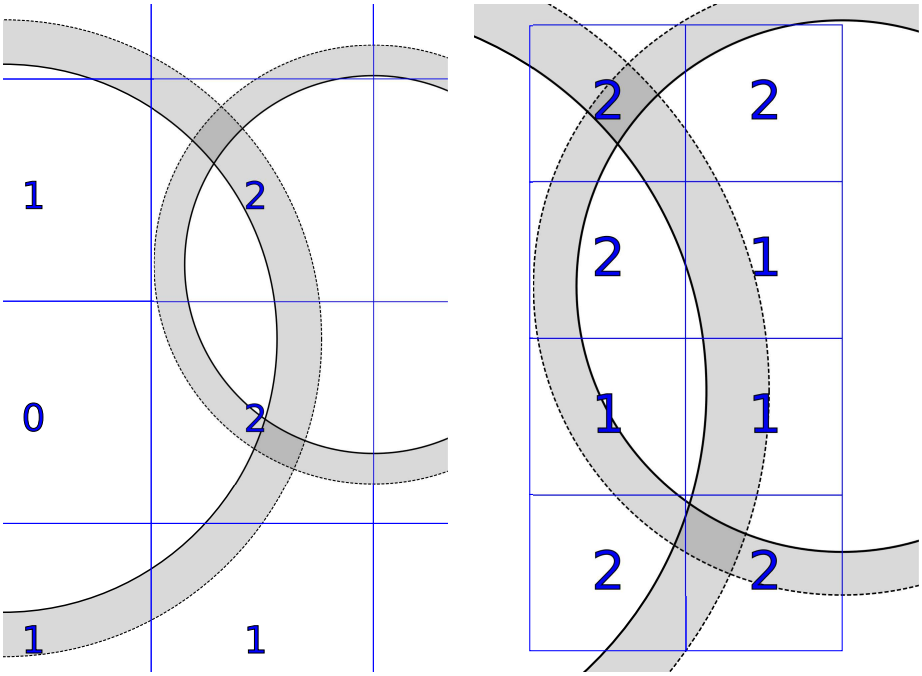
Listing 17: Preliminary step of the implemented 1-median algorithm

```
1  FUNCTION int
2  weiszfeld_test_optimum_opt(const __m128 ptsx, const __m128 ptsy, const __m128
       weights, const int i) {
3      __m128 xi = _mm_load1_ps(&ptsx[i]); __m128 yi = _mm_load1_ps(&ptsy[i]);
4      __m128 dists = distance(xi, yi, ptsx, ptsy);
5
6      __m128 tmpx = weights * ((xi - ptsx) / dists);
7      __m128 tmpy = weights * ((yi - ptsy) / dists);
8      __m128 inf = VECTOR_BROADCASTF(INFINITY);
9      __m128 mask = _mm_and_ps(_mm_cmplt_ps(tmpx, inf), _mm_cmpgt_ps(tmpx,
           -inf));
10     tmpx = _mm_blendv_ps(_mm_setzero_ps(), tmpx, mask);
11     tmpy = _mm_blendv_ps(_mm_setzero_ps(), tmpy, mask);
12
13     tmpx = _mm_hadd_ps(tmpx, tmpx); tmpx = _mm_hadd_ps(tmpx, tmpx);
14     tmpy = _mm_hadd_ps(tmpy, tmpy); tmpy = _mm_hadd_ps(tmpy, tmpy);
15
16     float result = sqrtf((tmpx[0] * tmpx[0]) + (tmpy[0] * tmpy[0]));
17     return (result <= weights[i]) ? i : -1;
18 }
```

4.5 Algorithm III: Optimized Voting Based Location Estimation

Like Geolateration, *Optimized Voting Based Location Estimation* (VBLE-OPT) is an algorithm that has not been published by the authors yet. It is an optimized version of the *Voting Based Location Estimation* algorithm proposed by Liu et al. in [LND05]. Both algorithms project a regular grid with a defined cell length onto the playing area and vote the cells depending on whether they are intersect by the anchor circles. Afterwards, the algorithms recursively reduce the size of the cells and repeat the voting procedure for all highest ranked cells, until a certain minimum grid size is reached. The essential difference lies in the way the voting on cells is performed: Liu et al. defined the voted area to the anchor circle's perimeter broadened by some error threshold factor both to the outside and to the inside of the circle. Will et al., by contrast, set this circular disk to lie entirely on the outside of the circle, thereby responding to the fact that real-world distance measurements in general can only have positive errors, i.e., the measured distances will be too large rather than too small[1]. Besides, VBLE-OPT contains several optimizations that reduce the number of cells to be voted, yet these optimization do not influence the position estimation. As an illustration of the voting process, Figure 8 displays two iterations of VBLE-OPT. Here, the dashed circle line represents the measured distance, i.e., the outer radius of the candidate ring.

[1] As explained in Section 4.1, in real-world localization technologies, the distance is usually determined by measuring the round-trip time (RTT) of a signal (e.g., radio waves) travelling between the anchor node and the client. Multiplying this time with the (fixed) speed of the signal yields the estimated distance. Since obstacles (e.g., walls) in the signal's path can only decelerate its speed but not accelerate it, it is effectivly impossible to obtain too short RTTs, assuming the used clock is accurate.



(a) First iteration (section) (b) Second iteration

Figure 8: Voting example

4.5.1 Functionality

As a preparative step, VBLE-OPT starts with the calculation of a minimum bounding rectangle that covers all anchors and their corresponding circles. The*candidate ring* of each anchor is defined as a circular disk around the anchor position whose outer and inner radii are the measured distance and this distance reduced by the error threshold, respectively. The bounding rectangle is then divided into square cells with a side length that is initially defined as 40 per cent of the bounding rectangle's shorter side. the target cell side length is defined as 5 per cent of the bounding rectangle's shorter side. The subsequent recursive main part of VBLE-OPT functions as follows:

- Iterating over the anchors, it first determines the outer test region of each anchor, which is the bounding box that completely covers the anchor's candidate ring, and the inner test region, which is defined by a number of cells that lie inside the candidate ring but do not intersect it.
- It then tests each cell contained in the outer test region but not in the inner test region for whether they are intersected by the candidate rings. For this purpose, maximum and minimum distances between the anchor position and the current cell's boundary are calculated and compared to the inner and outer radii of the candidate ring. In order to obtain these maximum and minimum distances, the algorithm determines the geometric sector of the anchor relative to the cell. See Figure 9 for an illustration of the sectors. In this example, the anchor position lies in sector 8, hence the minimum distance is to be found between

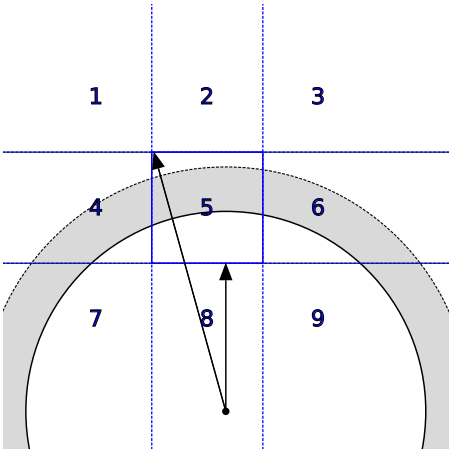


Figure 9: Sector determination

the anchor and the bottom side of the cell, whereas the maximum distance is between the anchor and the upper left corner.

- If the cell is intersected by the candidate ring of the current anchor, its voting score is incremented. Along the way, the algorithm keeps track of the current maximum score as well as the bounding rectangle that covers all cells that have been voted that score.
- After all anchors have been processed, the bounding rectangle containing all highest-ranked cells is used as the grid area of the following iteration, with the grid's cell side length being halved.

When the grid has reached the desired granularity, the centers of the highest-scored cells are accumulated and their center of mass is returned as the position estimation. For reference, the basic outline of the algorithm is shown in Listing 18.

Listing 18: Outline of the VBLE-OPT algorithm

```
while (grid_size > min_grid_size) {

  // Initialize voting array.
  // Set each cell's score to zero.

  for(int anchor = 0; anchor < count; anchor++) {

    // Calculate outer and inner test regions
    // (xMin, xMax, yMin, yMax)

    for(int x = xMin; x < xMax; x++) {
      for(int y = yMin; y < yMax; y++) {

        // Skip cell if it is contained in inner test region.

        // Determine sector of anchor relative to cell.

        // Calculate minimum and maximum distance between
        // the anchor and the cell's boundary.

        // Using these distances, test whether candidate ring
        // overlaps with cell. If so, increase the current cell's
        // score in the voting array.
      }
    }
  }
}

// Return center of mass of highest-ranked cells.
```

4.5.2 Optimizations

Despite its apparent simplicity, the VBLE-OPT algorithm was difficult to optimize due to its many code paths and irregular performing. For example, in the algorithm's inner two loops, which iterate over the cells to be voted on by their x- and y-coordinate, the loop counters are bounded by the calculated outer test region. Additionally, the inner-most loop may be cancelled by a `break` statement in case the current cell is contained in the inner test region. It is therefore hardly possible to know the number of iterations of these loops for a particular anchor beforehand, and it is also doubtful that this number is similar over multiple runs of the algorithm (i.e., multiple distance measurements for the same anchor). As a result, although most of the loop body can be vectorized easily, a number of blacklists need to be maintained in order to reflect the varying number of loop iterations.

Complete vectorization. These blacklists indicate whether a vector element (i.e., a run of the algorithm) has already passed the loop limits, in other words, whether its whole test region has already been processed. Fortunately, since the main portion of code contained in the loop body has no outside effects, the blacklists need to be examined only before updating the voting scores. My first attempt at optimizing VBLE-OPT consisted of a major rewrite of the algorithm that targeted at vectorizing it as a whole. I used SSE4 integer intrinsics to manipulate the loop counters in parallel. As a consequence, this prevents the implementation from making use of the AVX unit of recent processors, as AVX does not include integer instructions until now. Listing 19 displays an extract from the vectorized implementation of VBLE-OPT.

Listing 19: Simplified code of the vectorized implementation of VBLE-OPT

```
1  __m128i j = xMin; // xMin is a __m128i, too.
2  while (1) {
3      // Break condition: All j elements have passed their corresponding
4      // xMax values.
5      // We use this to blacklist those vector elements where j has passed the
6      // xMax value for the current iteration.
7      VECTOR jgexMax = VECTOR_CMPGE(_mm_cvtepi32_ps(j), _mm_cvtepi32_ps(xMax));
8      if (VECTOR_TEST_ALL_ONES(jgexMax))
9          break;
10
11     __m128i k = yMin;
12     while (1) {
13         // Same here.
14         VECTOR kgeyMax = VECTOR_CMPGE(_mm_cvtepi32_ps(k),
               _mm_cvtepi32_ps(yMax));
15         if (VECTOR_TEST_ALL_ONES(kgeyMax))
16             break;
17
18         // [Skipped: Calculate dMax/dMin distances.]
19
20         // Merge blacklists to guard the voting process.
21         VECTOR blacklist = VECTOR_OR(jgexMax, kgeyMax);
22
23         // Calculate voting array indices.
24         __m128i scoresIdx = _mm_mullo_epi32(j, yLength + _mm_set1_epi32(1)) +
               k;
25
26         // Test if candidate ring overlaps with cell.
27         VECTOR sx = VECTOR_AND(VECTOR_CMPLE(dMin, anchors[i].ro),
               VECTOR_CMPGE(dMax, anchors[i].ri));
28         VECTOR sx = VECTOR_AND(one, VECTOR_ANDNOT(blacklist, sx));
29
30         // Increase cell's scores if candidate ring overlaps.
31         scores[scoresIdx[0]][0] += (int) sx[0];
32         scores[scoresIdx[1]][1] += (int) sx[1];
33         scores[scoresIdx[2]][2] += (int) sx[2];
34         scores[scoresIdx[3]][3] += (int) sx[3];
35
36         // [Skipped: Maintain highest-ranked cell.]
37
38         k += _mm_set1_epi32(1);
39     }
40     j += _mm_set1_epi32(1);
41 }
```

Line 7 shows the creation of the first blacklist, which masks vector elements for which the loop counter `j` has already passed the corresponding `xMax` limit. The `_mm_cvtepi32_ps` instruction, which converts 4 packed integer values stored in an `xmm` register into 4 packed floats, fills the gap for the integer "greater or equal" comparison instruction that is still missing in SSE4. Lines 23–33 mark the voting procedure of VBLE-OPT: First, the array indices of the current cells are calculated using vector operations such as `_mm_mullo_epi32`, which multiplies two `xmm` registers filled with 4 signed integers. Based on the calculated minimum and maximum distances and the candidate ring (`anchor[i].ri` and `anchor[i].ro`), it is then decided which of the current cells should be voted for (l. 27). In line 28, the voting score increment of these cells is set to 1, but only if they were not blacklisted before, in which case it is set to zero. Finally, the score of each cell is updated. The latter also constitutes one of the major bottlenecks of the algorithm, since

the `scores` array needs to be accessed in a nonparallel and totally random way.
Not shown in Listing 19 is a lengthy code section that determines the anchor's geometric sector relative to the cells, which is necessary for calculating the anchor's minimum and maximum distance to the cell boundary (refer to Figure 9). The original implementation contained a long sequence of conditional statements that determined the sector by comparing the anchor's coordinates to the corner points of the current cell. In the SSE implementation, I replaced these conditionals with a series of `blendvps` instructions that blended the distances to the `dMax` and `dMin` vectors based on the results of a large number of vector comparisons. Additionally, since the original code left the sector determination once the proper sector was found (using a chain of `if-else-if-...` statements), it was necessary to maintain a bitmask that indicated which vector elements have been successfully processed and incorporate this bitmask into the blending instructions. In sum, the vectorized implementation unconditionally calculated every possible distance value, used plenty of comparisons and `blendvps` instructions, and required additional logical operations for maintaining the bitmask. By contrast, the scalar implementation required only a minimum number of arithmetic operations and up to 9 comparisons, in addition to a conditional jump. In the end, my initial approach at vectorizing VBLE-OPT turned out to perform about 1.5 times slower than the original implementation.

Minor optimizations. As a consequence of this initial failure, I started over with a clean version of the original code, this time trying to optimize smaller, isolated parts of the algorithm. The implementation, showing much potential for code restructuring and better memory management, was benefitted by the following measures:

- The voting array was dynamically allocated using `malloc`. Changing this to stack allocation measurably improved performance.
- Several constant values were repeatedly calculated in the inner loops. I modified the implementation to precalculate these values (e.g., the candidate rings of the anchors and the inner test region) in advance.
- To test whether a cell is overlapped by a candidate ring, the original implementation used euclidean distances. In order to save several square root instructions, I changed this to use squared euclidean distances instead.
- As a remarkable example of the *locality of reference* principle, I introduced the `anchor_info_t` struct that contains the coordinates of an anchor as well as its candidate ring and inner test region and thus keeps these values close together in memory. Since the anchors are processed iteratively, this considerably reduced the number of cache misses in the inner loops.
- For the final position estimation, the original implementation calculated the center of mass of the centers of the highest-ranked cells. In order to calculate these center positions, it added half of the grid cells' side length to the cells' coordinates, which point to the top-left corner of the cells. I deferred this "centering" until after the center of mass has been calculated, thereby saving a number of addition operations.

These combined measures led to a moderate improvement of the overall performance of the algorithm. However, the breakthrough results I had expected failed to materialize.

Sector determination. Later, while trying to reduce the number of branch mispredictions resulting from conditionals contained in the sector determination section, I found a mathematical approach to calculate the minimum and maximum distances that does not require any conditional statements. Although this again is an algorithmic optimization, I decided to include it in the algorithm as it drastically improved performance. Since it is not obvious to understand from the code (see Listing 20), I will give a brief explanation of this approach in the following, for later reference.
First, we need to calculate the point on the cell boundary that is closest to the anchor. This is a simple matter of "clamping" the anchor coordinates to the cell's vertical and horizontal extents, where clamping is defined as `clamp(v, min, max) = (v < min) ? min : ((v > max) ? max : v)`. Afterwards, we use vector mathematics to reach the most distance point, i.e., we travel along the longer section of the cell side that the closest point lies on and travel an entire cell side in the orthogonal direction (ll. 6–7). In lines 7–8 we construct the vector pointing to the closest point, however, since the `farXoffset` and `farYoffset` values are always positive, we mirror it onto the first quadrant by using absolute values (where figuratively the anchor is the coordinate system's origin). The vector pointing to the most-distant point is created by adding the offsets to the closest point vector (ll. 10–11). Finally, the squared euclidean distances are calculated in lines 12–13.
After I discovered this alternate solution to the minimum/maximum distance problem, I embedded it into the initial fully-vectorized implementation. Unexpectedly, this resulted in a significant performance boost, up to the point where this version largely outperformed the optimized scalar implementation. The final optimized version consists of the vectorized implementation enhanced with the smaller optimizations mentioned above. However, the achieved speed-up relies on the similarity of the input parameters provided by the engine. My experiments suggest that in the majority of cases, the number of loop iterations in the main loop is constant over all vector elements. This again may change with other error models being added to LS^2.

Listing 20: Sector determination using vector mathematics

```
1  // Determine minimum and maximum distance to cell boundary.
2  // Note that (x,y) denotes the top-left corner of the cell and
3  // L is its side length.
4  float closestX = CLAMP(anchors[i].x, x, x+L);
5  float closestY = CLAMP(anchors[i].y, y, y+L);
6  float farXoffset = max(closestX - x, (x+L) - closestX);
7  float farYoffset = max(closestY - y, (y+L) - closestY);
8  float minDistX = abs(closestX - anchors[i].x);
9  float minDistY = abs(closestY - anchors[i].y);
10 float maxDistX = minDistX + farXoffset;
11 float maxDistY = minDistY + farYoffset;
12 float dMinSqr = minDistX * minDistX + minDistY * minDistY;
13 float dMaxSqr = maxDistX * maxDistX + maxDistY * maxDistY;
```

4.6 Evaluation

In the following, I will evaluate the outcome of my optimizational work. Using experimental benchmark data, I will discuss the impact of the optimizations on the three algorithms and reason about the influence of anchor positions on each algorithm's performance. I will also debate the statistical significance of these benchmarks.

4.6.1 Approach

As mentioned in Section 4.2.2, the following data has been collected using my custom-made *benchlat* utility. For this final benchmark, the algorithms were executed 500 times each, using 3, 4, and 5 anchors for AML and VBLE-OPT, and only 3 anchors for Geo3, which simply discards additional anchors. Regarding the selection of anchor positions, I decided to mainly choose points close to distinct borders of the playing field, aiming at a near uniform distribution of the area of the shapes formed by the anchors, and at avoiding having a bias towards small, centered shapes that would arise from a uniform distribution of points. This can in general be considered more realistic for localization scenarios. Though, since the input parameters are discretized to integer simulation units by the engine, this constraint also resulted in a lower bound on the area of the anchor shape. For example, one minimal triangle would be formed by three points lying as close as possible to one border of the playing area without being colinear, for example, the points (1|0), (0|500), and (1|1000). This triangle still has an area of 500 square simulation units.
After each algorithm's run, the original implementation of the algorithm was executed using the same set of anchors, in order to obtain a reference runtime. Both the reference and the optimized runtimes were reduced by the constant overhead imposed by the engine itself, which is mainly composed by the time it takes the engine to calculate the real distances and to generate the random measuring errors. In order to estimate this overhead, I executed the *const* algorithm shipped with LS^2, which does nothing but return the input parameters, a thousand times and calculated the average of the runtimes. The resulting average overhead is 0.06603 seconds.
The achieved speed-up of each anchor and algorithm combination was calculated using Formula 1 shown below, where $t_{o,i}$ denotes the optimized runtime and $t_{r,i}$ is the reference runtime. The average speed-up was derived using the usual formula for the arithmetic mean that is displayed in Formula 2.

$$s_i = \frac{t_{r,i}}{t_{o,i}} \tag{1}$$

$$\texttt{average} = \frac{1}{n} \cdot \sum_{i=1}^{n} s_i \tag{2}$$

The benchmarks were run on a quad-core Intel Xeon E31245 CPU with a clock frequency of 3.30 GHz, which had access to a total of 8 GB main memory. Because availability of this system was temporally limited, the number of algorithm iterations for each position needed to be reduced to 40, though, as execution runtimes are linearly dependent on the number of iterations (refer to Section 4.1, this does not compromise the significance of the benchmark results. The framework was configured to always use uniformly distributed errors. This, by contrast, limits the universality of the benchmark results to some degree. However, at the time of my coding it was the only error model implemented in LS^2. The application was compiled using the GNU compiler collection, version 4.6.3, with compiler optimization set to `-O3`.

4.6.2 Discussion

To begin with, all benchmark runs have delivered a positive result, i.e., the performance of each algorithm has been improved no matter where the anchors were placed. Table 2 shows the average speed-up calculated over all benchmark runs for each algorithm for 3, 4, and 5 anchors, providing a convenient overview of how much the algorithms benefitted from the optimization. AML boasts the best results with an average speed-up of 2.7 to 3.0, followed by Geolateration and VBLE-OPT.

Table 2: Average speed-ups

	Average speed-up		
Algorithm	3 anchors	4 anchors	5 anchors
AML	3.30	3.34	3.36
GEO3	2.20	N/A	N/A
VBLE-OPT	1.68	1.77	1.75

Most prominently, the AML speed-ups seem to increase linearly, which was affirmed by (less numerous) experiments I conducted with 6 to 8 anchors. This is likely caused by the growing influence of the *refinement* step, that could be perfectly vectorized due to the absence of conditional statements and thus has a considerable advantage over the scalar implementation. Using fewer anchors, the *first intersection* step becomes the prevalent factor limiting the overall performance. However, as mentioned in the introduction, the theoretical speed-up of a SSE-based optimization is the number of vector elements, which is 4. I therefore suspect this increase to slowly decline with additional anchors until the speed-up factor has reached its peak, where it will probably stagnate or maybe start to decline due to other effects such as memory bandwidth limits.

In general, it is arguable whether these average speed-ups can be used to evaluate the success of my optimizational work. The algorithms showed remarkable differences in the distribution of the test data, which can be seen in the box-and-whisker diagram in Figure 10. Whereas speed-ups of AML and VBLE-OPT have a small variance, Geolateration has a considerably larger dispersion with its interquartile range spreading from around 2.05 to 2.35. I expect this to a be consequence of the many early-out conditionals contained in Geo3, which in general lead to large variances in the algorithm runtimes. Some manual investigation into the anchor placements for speed-ups both from the lower and from the upper range of results suggested that input anchors forming a large triangle (e.g., anchors lying close to distinct corners of the playing area) led to higher speed-ups, whereas small, acute-angled triangles resulted in lower speed-ups. In the latter case, the optimizational step 4, which tests whether the minimum perimeter triangle is contained in the triangle formed by the anchors, is more likely to fail and therefore the remaining parts are executed more often. This also shows that the overall speed-up of Geo3 is not cumulative over the performance improvements of the various sections: Whereas the optimization of step 5, which calculates the geometric median of the final triangle, proved highly beneficial in my experiments, the overall speed-up of the optimization is higher in cases where it is executed less frequently.

To put some visual emphasis on the distribution of the speed-ups with respect to the spread of the runtimes, Figures 11 and 12 display scatterplots that relate the reference runtimes to the corresponding optimized runtimes, where each point represents a single pair of runtimes. The position of points relative to the x-axis shows the general variance of reference runtimes for the given 500 random benchmark runs. Additionally, the scattering of points around the line, which interpolates the expected optimized runtimes using the calculated average speed-up (i.e., it is a plot of the formula $y = \frac{x}{\texttt{average speed-up}}$), illustrates the variance of the speed-ups. For example, to verify what has been said above, the upper right region of Figure 11(a) shows that higher runtimes of Geo3 had an lower-than-average speed-up.

With regards to Adapted Multilateration, although the speed-up variance is lower than in the case of Geo3, the data has only a weak empiric correlation[1] of 0.172, i.e., long reference runtimes

[1] For an introduction to the sample correlation coefficient, see, for example, [Ros04, p. 33ff].

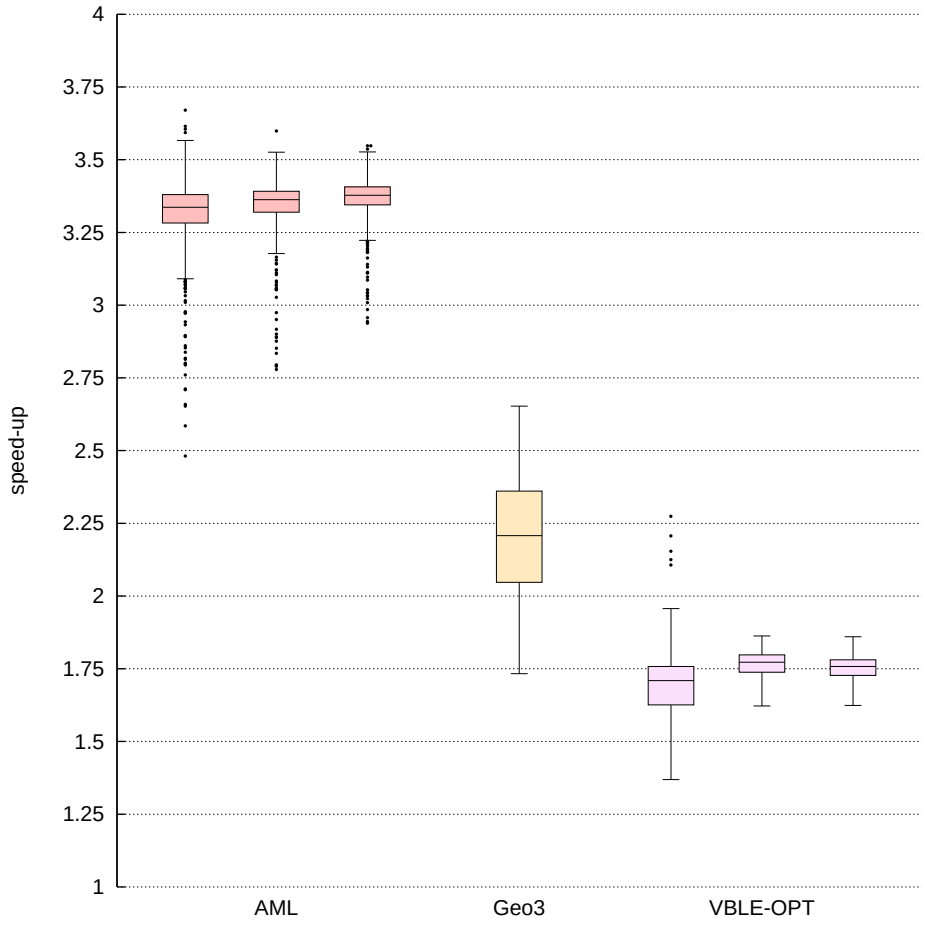
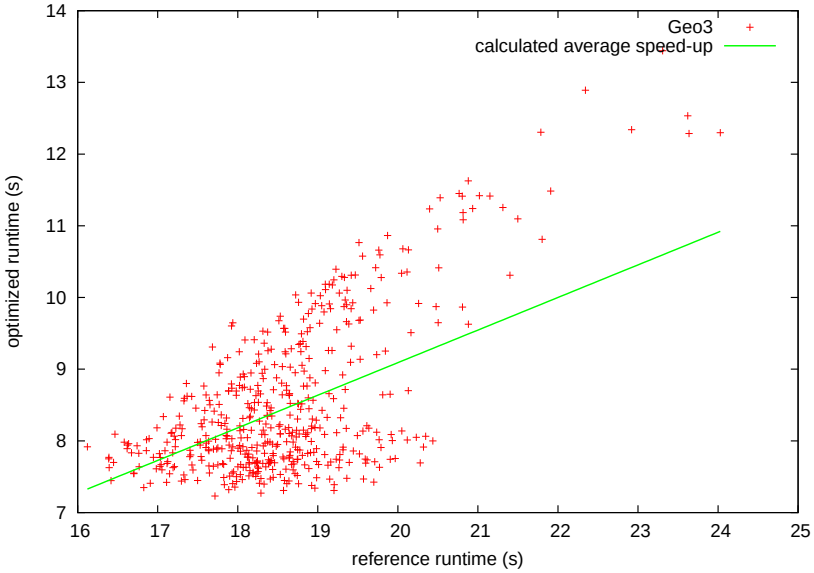
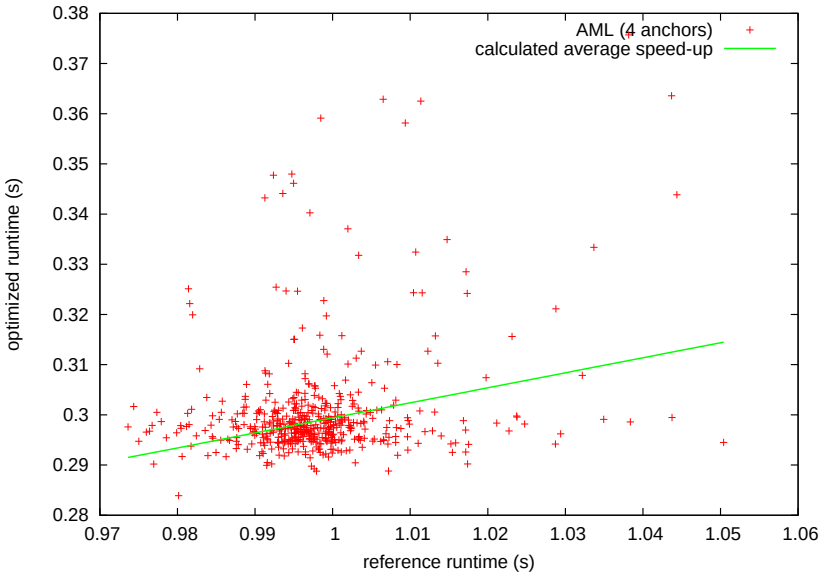


Figure 10: Distribution of speed-ups



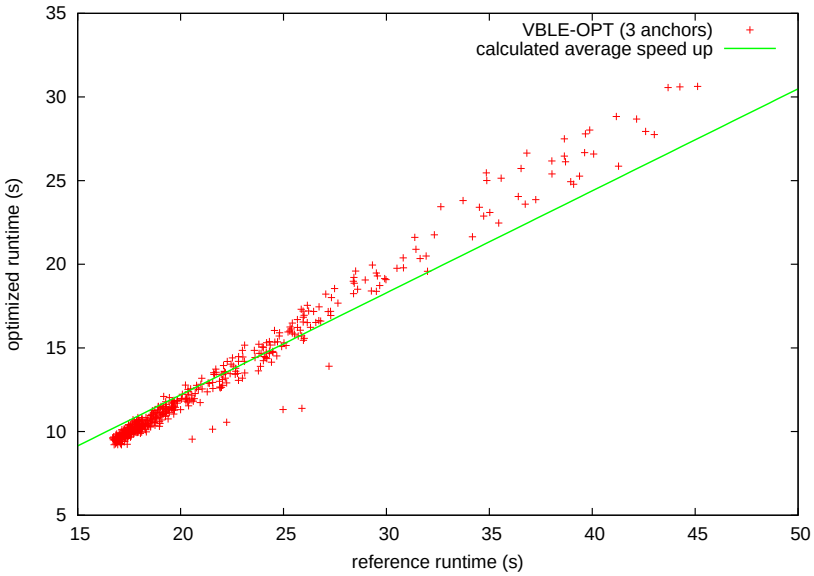
(a) Geo3



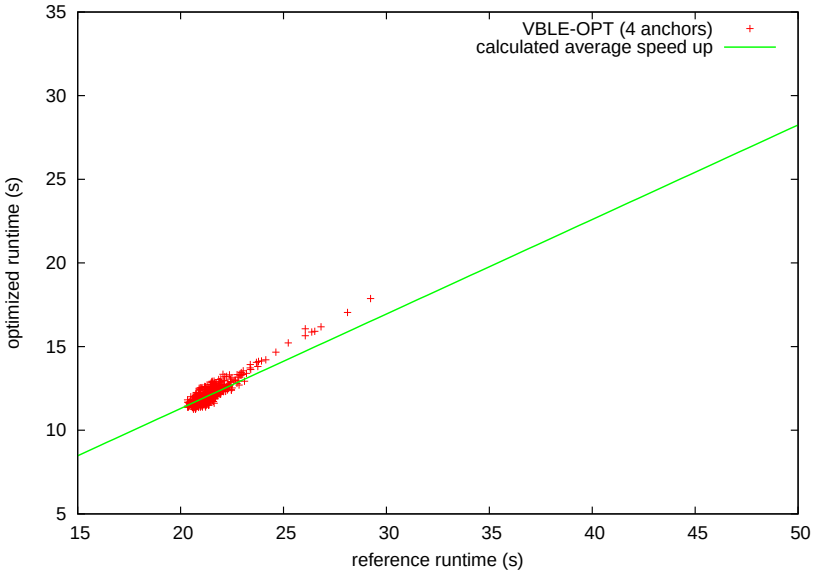
(b) AML (4 anchors)

These scatterplots show the reference runtimes in relation to the corresponding optimized runtimes, so each point actually marks two measured values. The line is an interpolation of the *calculated average speed-up*, i.e. it can be read as, "Based on the average speed-up, one can expect a run of the algorithm that took x seconds in the original implementation to take y seconds in the optimized version".

Figure 11: Scatterplots of benchmark results



(a) 3 anchors



(b) 4 anchors

Figure 12: Benchmark results of VBLE-OPT

did not necessarily result in longer optimized runtimes, and vice versa. This may result from the *first intersection* step occasionally needing more iterations to find circle intersections for all vector elements, in which case the benefits of the vectorization are reduced by the increased processing time spent at the vector comparisons and blending instructions Yet in other runs, the vectorization has larger impacts and runs that were previously slow turned out to have considerable shorter runtimes in the optimized implementation. The scatterplot displayed in Figure 11(b) shows that the speed-up is quite often lower than average (i.e., above the line), yet there seems to exist no connection between the speed-ups and the original runtimes.
In view of the benchmarks of VBLE-OPT, I would like to point out that the runtimes that were measured for sets of 3 anchors differed substantially from the runtimes of benchmarks using 4 or 5 anchor (or more). Theoretically, the performance of VBLE-OPT should be almost constant with respect to the number of anchors, as the number of cells in the grid is independent of the anchor position and there are no early-out conditionals and the like. Still, when the algorithm was parametrized with 3 anchors, the runtimes varied a lot, yet only towards above the overall average (see Figure 12). These slower runs also showed a slightly worse speed-up factor, which resulted in a larger speed-up variance, as depicted in the box-and-whiskers diagram. The reason for these varying runtimes is still unclear to me and could be investigated in future research.
In summary, it can be stated that the outcome of the benchmarks on the whole has confirmed the positive impact of the applied SSE optimization on the performance of each algorithm. However, the methodic approach that was used to acquire these numbers contained several flaws, as described in Section 4.6.1. Since there was a lower bound on the area of the shape formed by the anchors, a certain range of possible input shapes has not been tested. In the case of Geolateration, where the benchmarks suggest that smaller triangles have led to lower speed-up factors, this may have influenced the average speed-ups. Still, as these minimum area cases are generally not very likely, the overall tendency of the benchmarks should be correct.

5 Conclusion

In this thesis, I have shown how real-world algorithms can be optimized using SSE vector intrinsics. I described the theoretical background in Section 3 and conducted a case-study on the LS^2 simulation engine in Section 4.
The results of this case-study, as discussed in Section 4.6, have been highly rewarding. Especially the *Adapted Multilateration* algorithm has extremely benefited from the applied vectorization, with the achieved average speed-up of 3.3 being close to the theoretical maximum speed-up of 4. In other words, the average runtimes of the algorithm could be reduced to a third of the runtimes of the original implementation. Likewise, the performances of both VBLE-OPT and Geolateration, in spite of these algorithms being far more complex with regard to the control flow, could be drastically improved.
From the outcome of the case-study, one can further observe two simple facts: First, although compiler optimization may be sufficient for a decent performance of the average application, manual code optimization may still yield significant performance gains in computationally intensive applications. Today, modern compilers equipped with auto-vectorization features are able to automatically vectorize simple loops using SSE instructions, yet they lack the comprehensive view over an algorithm that is needed to identify vectorization opportunities as complex as demonstrated in the LS^2 optimization. Auto-vectorization will remain an attractive research topic in the future and it will be interesting to see how auto-vectorization techniques will be adapted to cope with more complex situations. However, it is worth mentioning, that in some situations the compiler will never be able to decide whether vectorization is a valid option, as it

may require changes to the algorithm's functionality.
Second, the results show vividly that the capabilities of modern processors — especially the integrated SIMD technology — are not at all used to their full potentials. Even though the LS^2 application had been optimized already and was compiled with compiler optimization set to its highest level, I was able to *triple* the performance of AML on the same hardware. To emphasize this point even more, the same result could have been achieved by installing two additional processors of the same kind into the system, disregarding any memory bandwidth limitations.
Certainly, the potential impact of SSE optimization largely depends on the inherent data-parallelism of a particular algorithm. If an algorithm contains overly many conditional jumps or processes data in totally random patterns, vectorization is difficult to accomplish and is very likely to have no advantageous effects on the algorithm's performance. Similarly, algorithms whose performance is entirely limited by memory bandwith will rarely benefit from vectorization. The LS^2 engine, by contrast, has proven an ideal application to be enhanced by SSE optimization, as its main functionality, the endlessly repeated position estimation using lateration algorithms, could be perfectly parallelized. Additionally, since the implemented algorithms are not subjected to future changes, the increased complexity and reduced readability that come along with SSE optimization are not as problematic as they might be for other applications, for which it should be decided on a case-by-case basis whether SSE vectorization as well as manual optimization in general is worth the effort.

References

[AMD12] AMD. Software Optimization Guide for AMD Family 15h Processors. `http://support.amd.com/us/Processor_TechDocs/47414_15h_sw_opt_guide.pdf` [last accessed: May 4, 2014], 2012.

[Bou97] Paul Bourke. Intersection of two circles. `http://local.wasp.uwa.edu.au/~pbourke/geometry/2circle/` [last accessed: May 4, 2014], 1997.

[Dre07] Ulrich Drepper. What Every Programmer Should Know About Memory. `http://people.redhat.com/drepper/cpumemory.pdf` [last accessed: May 4, 2014], 2007.

[Fog11a] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. `http://www.agner.org/optimize/instruction_tables.pdf` [last accessed: May 4, 2014], 2011.

[Fog11b] Agner Fog. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms. `http://www.agner.org/optimize/optimizing_cpp.pdf` [last accessed: May 4, 2014], 2011.

[GG08] Jason Garrett-Glaser. Cacheline splits, aka intel hell. `http://x264dev.multimedia.cx/archives/8` [last accessed: May 4, 2014], 2008.

[HZS09] Guy Ben Haim, Victoria Zhislina, and Sagi Schein. Optimization of Image Processing Algorithms: A Case Study. `http://software.intel.com/en-us/articles/optimization-of-image-processing-algorithms-a-case-study/` [last accessed: May 4, 2014], 2009.

[Int11] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. `http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf` [last accessed: May 4, 2014], 2011.

[Int12] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, and 3C. `http://download.intel.com/products/processor/manual/325462.pdf` [last accessed: May 4, 2014], 2012. Volume 2, Appendix C.

[KEO09] Gulnur Selda Kuruoglu, Melike Erol, and Sema Oktug. Localization in wireless sensor networks with range measurement errors. In *Proceedings of the 2009 Fifth Advanced International Conference on Telecommunications*, pages 261–266, 2009.

[Knu74] Donald E. Knuth. Structured Programming with go to Statements. *Computing Surveys*, 6:268, 1974.

[Knu98] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[Lei09] Felix von Leitner. Source Code Optimization. `http://www.linux-kongress.org/2009/slides/compiler_survey_felix_von_leitner.pdf` [last accessed: May 4, 2014], 2009.

[Lir09] LiraNuna. SSE intrinsics in popular compilers. `http://www.liranuna.com/sse-intrinsics-optimizations-in-popular-compilers/` [last accessed: May 4, 2014], 2009.

[LND05] Donggang Liu, Peng Ning, and Wenliang Kevin Du. Attack-Resistant Location Estimation in Sensor Networks. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[MH99] Miles J. Murdocca and Vincent P. Heuring. *Principles of Computer Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[Ros04] Sheldon M. Ross. *Introduction to Probability and Statistics for Engineers and Scientists, Third Edition*. Academic Press, 3 edition, 2004.

[SJV06] Asadollah Shahbahrami, Ben Juurlink, and Stamatis Vassiliadis. Performance Impact of Misaligned Accesses in SIMD Extensions. In *Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2006)*, pages 334–342, 2006.

[Str02] Cort Stratton. Optimizing for SSE: A Case Study. `http://www.cortstratton.org/articles/OptimizingForSSE.php` [last accessed: May 4, 2014], 2002.

[Ton12] Tuomas Tonteri. A practical guide to using SSE with C++. `http://sci.tuomastonteri.fi/programming/sse` [last accessed: May 4, 2014], 2012.

[WHK12] Heiko Will, Thomas Hillebrandt, and Marcel Kyas. The FU Berlin Parallel Lateration-Algorithm Simulation and Visualization Engine. In *Proceedings of the 9th Workshop on Positioning, Navigation and Communication 2012*, 2012.

[WP09] E. Weiszfeld and Frank Plastria. On the point for which the sum of the distances to n given points is minimum. *Annals of Operations Research*, 167:7–41, 2009.

List of Figures

List of Tables

List of Listings

Appendices

A Three `memset` implementations

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <smmintrin.h>

void tim(const char *name, float *arr,
         int length, void (*func)(float*, int, float)) {
    struct timespec t1, t2;
    clock_gettime(CLOCK_REALTIME, &t1);
    func(arr, length, 5.0f);
    clock_gettime(CLOCK_REALTIME, &t2);
    printf("%s : %f s.\n", name,
        (t2.tv_sec - t1.tv_sec) +
        (float) (t2.tv_nsec - t1.tv_nsec) / 1000000000);
}

void memset1(float *arr, int length, float value) {
    for(int i = 0; i < length; i++) {
        arr[i] = value;
    }
}

void memset2(float *arr, int length, float value) {
    __m128 buf = _mm_set1_ps(value);
    for(int i = 0; i < length; i += 4) {
        _mm_store_ps(&arr[i], buf);
    }
}

void memset3(float *arr, int length, float value) {
    __m128 buf = _mm_set1_ps(value);
    for(int i = 0; i < length; i += 4) {
        _mm_stream_ps(&arr[i], buf);
    }
    _mm_sfence();
}

int main() {
    const int CACHE_LINE = 64;
    int length = CACHE_LINE * 1024 * 1024;

    // Allocate 64 MB of memory aligned to a cache line.
    float *arr;
    posix_memalign((void**) &arr, CACHE_LINE,
                   length * sizeof(float));
    // Execute memset once to make sure the memory is allocated.
    memset1(arr, length, 0.0f);

    // Benchmark.
    tim("scalar", arr, length, memset1);
    tim("store", arr, length, memset2);
    tim("stream", arr, length, memset3);

    free(arr);
    return 0;
}
```

B Original implementation of circle_get_intersection

```
/**
 * Returns the intersections of two circles.
 * @ret: the number of intersection [0,1,2]
 */
FUNCTION int
circle_get_intersection (float  p1x, float  p1y, float  p2x, float  p2y,
    float r1, float r2, float* retx, float* rety) {
    float d = distance_s(p1x,p1y,p2x,p2y);

    // no solutions, the circles are separate || the circles are coincident
        || no solutions because one circle is contained within the other
    // => infinite number of solutions possible
    if (r1+r2 < d || fabs(r1-r2) > d || d == 0) {
        return 0;
    }

    float a = (r1*r1 - r2*r2 + d*d) / (2.0 * d);
    float v = r1*r1 - a*a;
    float h = sqrtf(v);

    float dx = (p2x - p1x) / d;
    float dy = (p2y - p1y) / d;
    float p3x = p1x + a * dx;
    float p3y = p1y + a * dy;

    dx *= h;
    dy *= h;
    float p4x = p3x + dy;
    float p4y = p3y - dx;

    int count = (p4x == p3x && p4y == p3y) ? 1 : 2;

    retx[0] = p4x;
    rety[0] = p4y;
    if (count == 2) {
        p4x = p3x - dy;
        p4y = p3y + dx;
        retx[1] = p4x;
        rety[1] = p4y;
    }
    return count;
}
```

C AVX implementation of the Geo3 distance precalculation

```
union {
    VECTOR v;
    float f[8];
} distances[sum];
VECTOR xt = _mm256_load_ps(ix);
VECTOR yt = _mm256_load_ps(iy);
for(int i = 0; i < sum; i++) {
    VECTOR px = VECTOR_BROADCAST(&ix[i]);
    VECTOR py = VECTOR_BROADCAST(&iy[i]);
    distances[i].v = distance(px, py, xt, yt);
}
```