Extending C++ for Explicit Data-Parallel Programming via SIMD Vector Types

Dissertation

zur Erlangung des Doktorgrades

der Naturwissenschaften

vorgelegt beim Fachbereich 12

der Johann Wolfgang Goethe-Universität

in Frankfurt am Main

von

Matthias Kretz

aus Dieburg

Frankfurt (2015)

(D 30)

vom Fachbereich 12 der

Johann Wolfgang Goethe-Universität als Dissertation angenommen.

Dekan:	Prof. Dr. Uwe Brinkschulte
Gutachter:	Prof. Dr. Volker Lindenstruth
	Prof. Dr. Ulrich Meyer
	Prof. Dr.-Ing. André Brinkmann
Datum der Disputation:	16. Oktober 2015

CONTENTS

ABSTRACT vii

PUBLICATIONS ix

ACKNOWLEDGMENTS xi

I INTRODUCTION

1 SIMD 3

1.1 The SIMD Idea 3

1.2 SIMD Computers 5

1.3 SIMD Operations 6

1.4 Current SIMD Instruction Sets 8

1.5 The future of SIMD 10

1.6 Interaction With Other Forms of Parallelism 11

1.7 Common Optimization Challenges 12

1.8 Conclusion 14

2 LOOP-VECTORIZATION 15

2.1 Definitions 15

2.2 Auto-Vectorization 18

2.3 Explicit Loop-Vectorization 21

3 INTRODUCTION TO SIMD TYPES 23

3.1 The SIMD Vector Type Idea 23

3.2 SIMD Intrinsics 24

3.3 Benefits of a SIMD Vector Type 25

3.4 Previous Work on SIMD Types 26

3.5 Other Work on SIMD Types 27

II Vc: A C++ LIBRARY FOR EXPLICIT VECTORIZATION

4 A DATA-PARALLEL TYPE 31

4.1 The Vector<T> Class Template 32

4.2 SIMD Vector Initialization 35

4.3 Loads and Stores 40

4.4 Unary Operators 44

4.5 Binary Operators 45

4.6 Compound Assignment Operators 50

4.7 Subscript Operators 51

4.8 Gather & Scatter 52

4.9 Selecting the Default SIMD Vector Type 68
4.10 Interface to Internal Data 70
4.11 Conclusion 71
5 DATA-PARALLEL CONDITIONALS 73
5.1 Conditionals in SIMD Context 74
5.2 The Mask<T> Class Template 76
5.3 Write-Masking 84
5.4 Masked Gather & Scatter 87
5.5 Conclusion 88
6 VECTOR CLASS DESIGN ALTERNATIVES 89
6.1 Discussion 89
6.2 Conclusion 92
7 SIMDARRAY: VALARRAY AS IT SHOULD HAVE BEEN 93
7.1 std::valarray<T> 93
7.2 Vc::SimdArray<T, N> 94
7.3 Uses for SimdArray 95
7.4 Implementation Considerations 96
7.5 Mask Types 98
7.6 Conclusion 98
8 ABI CONSIDERATIONS 99
8.1 ABI Introduction 99
8.2 ABI Relevant Decisions in Vc 99
8.3 Problem 100
8.4 Solution Space 106
8.5 Future Research & Development 108
9 LOAD/STORE ABSTRACTIONS 109
9.1 Vectorized STL Algorithms 109
9.2 Vectorizing Container 111
9.3 Conclusion 111
10 AUTOMATIC TYPE VECTORIZATION 113
10.1 Template Argument Transformation 116
10.2 Adapter Type 118
10.3 Scalar Insertion and Extraction 123
10.4 Complete Specification 124
10.5 Current Limitations 125
10.6 Conclusion 126

III VECTORIZATION OF APPLICATIONS

11 NEAREST NEIGHBOR SEARCHING 129
11.1 Vectorizing Searching 130

11.2 Vectorizing Nearest Neighbor Search 134
11.3 k-d Tree 139
11.4 Conclusion 152
12 VC IN HIGH ENERGY PHYSICS 155
12.1 ALICE 155
12.2 Kalman-filter 166
12.3 Geant-V 167
12.4 ROOT 170
12.5 Conclusion 171

IV CONCLUSION

13 CONCLUSION 175

V APPENDIX

A WORKING-SET BENCHMARK 179
B LINEAR SEARCH BENCHMARK 183
C NEAREST NEIGHBOR BENCHMARK 187
D BENCHMARKING 193
E C++ IMPLEMENTATIONS OF THE K-D TREE ALGORITHMS 195
E.1 *INSERT* Algorithm 195
E.2 *FINDNEAREST* Algorithm 197
F ALICE SPLINE SOURCES 203
F.1 Spline Implementations 203
F.2 Benchmark Implementation 209
G THE SUBSCRIPT OPERATOR TYPE TRAIT 213
H THE ADAPTSUBSCRIPTOPERATOR CLASS 215
I SUPPORTING CUSTOM DIAGNOSTICS AND SFINAE 217
I.1 Problem 217
I.2 Possible Solutions 219
I.3 Evaluation 220

ACRONYMS 221
BIBLIOGRAPHY 223
INDEX 231
GERMAN SUMMARY 235

ABSTRACT

Data-parallel programming is more important than ever since serial performance is stagnating. All mainstream computing architectures have been and are still enhancing their support for general purpose computing with explicitly data-parallel execution. For CPUs, data-parallel execution is implemented via SIMD instructions and registers. GPU hardware works very similar allowing very efficient parallel processing of wide data streams with a common instruction stream.

These advances in parallel hardware have not been accompanied by the necessary advances in established programming languages. Developers have thus not been enabled to explicitly state the data-parallelism inherent in their algorithms. Some approaches of GPU and CPU vendors have introduced new programming languages, language extensions, or dialects enabling explicit data-parallel programming. However, it is arguable whether the programming models introduced by these approaches deliver the best solution. In addition, some of these approaches have shortcomings from a hardware-specific focus of the language design. There are several programming problems for which the aforementioned language approaches are not expressive and flexible enough.

This thesis presents a solution tailored to the C++ programming language. The concepts and interfaces are presented specifically for C++ but as abstract as possible facilitating adoption by other programming languages as well. The approach builds upon the observation that C++ is very expressive in terms of types. Types communicate intention and semantics to developers as well as compilers. It allows developers to clearly state their intentions and allows compilers to optimize via explicitly defined semantics of the type system.

Since data-parallelism affects data structures and algorithms, it is not sufficient to enhance the language's expressivity in only one area. The definition of types whose operators express data-parallel execution automatically enhances the possibilities for building data structures. This thesis therefore defines low-level, but fully portable, arithmetic and mask types required to build a flexible and portable abstraction for data-parallel programming. On top of these, it presents higher-level abstractions such as fixed-width vectors and masks, abstractions for interfacing with containers of scalar types, and an approach for automated vectorization of structured types.

The Vc library is an implementation of these types. I developed the Vc library for researching data-parallel types and as a solution for explicitly data-parallel programming. This thesis discusses a few example applications using the Vc library showing the real-world relevance of the library. The Vc types enable parallelization of search algorithms and data structures in a way unique to this solution. It shows the importance of using the type system for expressing data-parallelism. Vc has also become an important building block in the high energy physics community. Their reliance on Vc shows that the library and its interfaces were developed to production quality.

PUBLICATIONS

The initial ideas for this work were developed in my *Diplom* thesis in 2009:

> [55] Matthias Kretz. "Efficient Use of Multi- and Many-Core Systems with Vectorization and Multithreading." Diplomarbeit. University of Heidelberg, Dec. 2009

The Vc library idea has previously been published:

> [59] Matthias Kretz et al. "Vc: A C++ library for explicit vectorization." In: *Software: Practice and Experience* (2011). ISSN: 1097-024X. DOI: `10.1002/spe.1149`

An excerpt of Section 12.2 has previously been published in the pre-Chicago mailing of the C++ standards committee:

> [58] Matthias Kretz. *SIMD Vector Types*. ISO/IEC C++ Standards Committee Paper. N3759. 2013. URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3759.html`

Chapters 4 & 5 and Appendix I have previously been published in the pre-Urbana mailing of the C++ standards committee:

> [57] Matthias Kretz. *SIMD Types: The Vector Type & Operations*. ISO/IEC C++ Standards Committee Paper. N4184. 2014. URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4184.pdf`
>
> [56] Matthias Kretz. *SIMD Types: The Mask Type & Write-Masking*. ISO/IEC C++ Standards Committee Paper. N4185. 2014. URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4185.pdf`
>
> [60] Matthias Kretz et al. *Supporting Custom Diagnostics and SFINAE*. ISO/IEC C++ Standards Committee Paper. N4186. 2014. URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4186.pdf`

ACKNOWLEDGMENTS

For Appendix I, I received a lot of useful feedback from Jens Maurer, which prompted me to rewrite the initial draft completely. Jens suggested to focus on deleted functions and contributed Section I.3.

Part I

Introduction

1

SIMD

Computer science research is different from these more traditional disciplines. Philosophically it differs from the physical sciences because it seeks not to discover, explain, or exploit the natural world, but instead to study the properties of machines of human creation.

— Dennis M. Ritchie (1984)

This thesis discusses a programming language extension for portable expression of data-parallelism, which has evolved from the need to program modern SIMD[1] hardware. Therefore, this first chapter covers the details of what SIMD is, where it originated, what it can do, and some important details of current hardware.

1.1 THE SIMD IDEA

The idea of SIMD stems from the observation that there is data-parallelism in almost every compute intensive algorithm. Data-parallelism occurs whenever one or more operations are applied repeatedly to several values. Typically, programming languages only allow iterating over the data, which leads to strictly serial semantics. Alternatively (e.g. with C++), it may also be expressed in terms of a standard algorithm applied to one or more container objects.[2]

Hardware manufacturers have implemented support for data-parallel execution via special instructions executing the same operation on multiple values (Listing 1.1). Thus, a single instruction stream is executed but multiple (often 4, 8, or 16) data sets are processed in parallel. This leads to an improved transistors per FLOPs[3] ratio[4] and hence less power usage and overall more capable CPUs.

1 Single Instruction, Multiple Data
2 Such an algorithm can relax the iteration ordering and thus place additional restrictions on the semantics—in terms of concurrency—of the executed code. (cf. [36])
3 floating-point operations
4 The number of transistors required for instruction decoding and scheduling is large relative to the transistors needed for the arithmetic and logic unit. However, mainly the latter must be increased if

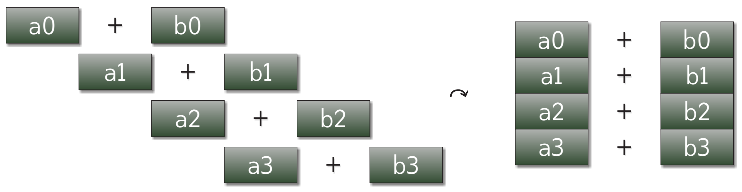


Figure 1.1: Transformation of four scalar operations into one SIMD operation.

```
; scalar multiply-add: 1 FLOP per instruction
mulss %xmm1,%xmm0
addss %xmm2,%xmm0

; SSE multiply-add: 4 FLOP per instruction
mulps %xmm1,%xmm0
addps %xmm2,%xmm0

; AVX multiply-add: 8 FLOP per instruction
vmulps %ymm1,%ymm0,%ymm0
vaddps %ymm2,%ymm0,%ymm0

; AVX with FMA multiply-add: 16 FLOP per instruction
vfmadd213ps %ymm1,%ymm0,%ymm2
```

Listing 1.1: Example of SIMD instructions in x86-64 assembly.

Figure 1.1 visualizes the conceptual idea of SIMD execution. Code that needs to execute four add operations on four different value pairs needs four instructions to execute the addition operations. Alternatively, the same result can be achieved with a single SIMD instruction that executes four addition operations on two vectors with four values each. In terms of operations per instruction, the SIMD variant is a factor four better. Since on most modern CPUs the scalar and the vector instruction execute equally fast this translates directly to a factor four higher operations per cycle throughput.

In the following, $\mathcal{W}_{\mathtt{T}}$ denotes the number of scalar values in a SIMD vector register of type `T` (the width of the vector). Thus, with SSE[6] or AltiVec $\mathcal{W}_{\mathtt{float}} = 4$ and with AVX[7] it is 8 (see Section 1.4). In addition, $\mathcal{S}_{\mathtt{T}}$ denotes `sizeof(T)`. It follows that $\mathcal{S}_{\mathtt{T}} \cdot \mathcal{W}_{\mathtt{T}}$ is the size of a SIMD vector of type `T` in Bytes.

a CPU[5] adds SIMD instructions or grows the SIMD width. Therefore, the total number of transistors grows by a fraction whereas the peak performance (in terms of operations per cycle) doubles.
6 Streaming SIMD Extensions
7 Advanced Vector Extensions

1.2 SIMD COMPUTERS

In the original Von Neumann architecture [66] from 1945 the processing unit of a computer consists of a "central arithmetical part", a "central control part", various forms of memory, and I/O. In all of the first computer designs registers and operations were meant to store and process only single (scalar) values. Thus, the ALUs[8] only processed one value (normally two inputs and one output) with a given instruction. This model later became known as SISD (according to Flynn [24]), because a *Single Instruction* processes a *Single Data* element.

In 1972, the ILLIAC IV [38] was the first array machine to introduce SIMD computing [78]. Array machines employ multiple processing units (PEs[9]) in parallel, which are controlled from the same instruction stream. They may support conditional execution via flags that enable/disable a selected PE. The PEs have some form of local memory and a local index to identify their position in the whole machine. Tanenbaum [78] describes the history and architecture of these machines in more detail. These machines became known as SIMD machines because a *Single Instruction* processes *Multiple Data* elements.

The modern implementation of SIMD machines are called vector machines. The name *vector* stems from the use of wide registers able to store multiple values. Associated *vector instructions* process all entries in such *vector registers* in parallel. For some time, vector instructions were only targeted at specific problems, such as multimedia processing. However, over the years many more instructions have been introduced, making the vector units more suited for general purpose computing.

Whereas array machines have a greater separation between the different PEs, vector machines use the same memory for the whole vector register. Additionally, there is no natural identification in vector machines that would resemble the local index of array machines. Vector instructions, in most hardware, calculate the result for all vector lanes[10] and have no means to selectively write to the destination register. Subsequent blend instructions (or a combination of `and`, `andnot`, `or`) can be used to achieve conditional assignment. Newer vector instructions (the Intel Xeon Phi and AVX-512) now support generic write-masking encoded into every instruction.

The rest of the document will focus on vector machines. Most considerations can be directly translated to a classical SIMD machine, though.

8 Arithmetic Logic Units
9 Processing Elements
10 A vector lane (or SIMD lane) identifies a scalar entry in a vector register. Most SIMD instructions execute the same operation per lane and do not read/write any data from/to lanes at a different offset in the register. In that sense they are comparable to the PEs of array machines.

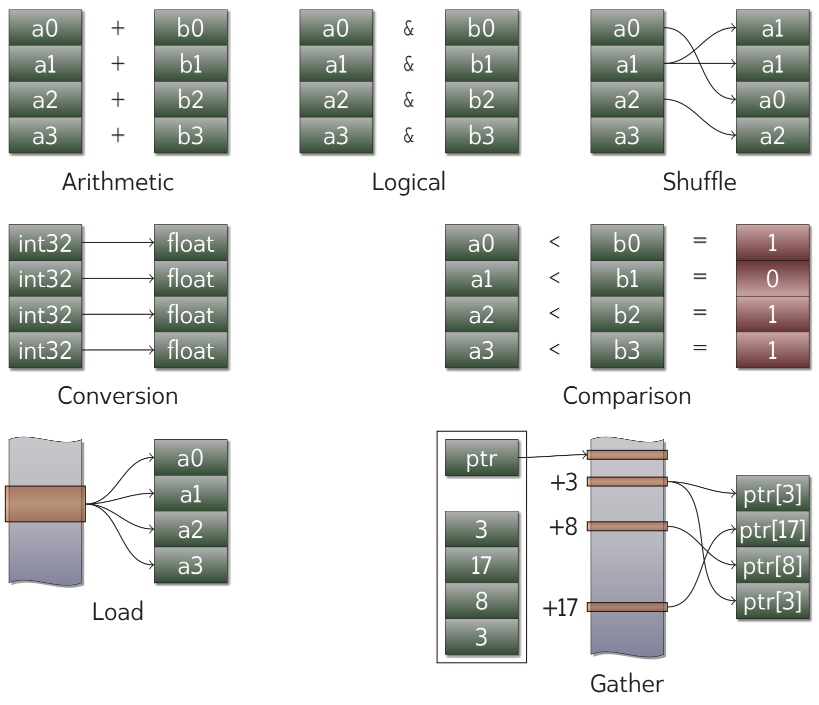


Figure 1.2: Overview over common SIMD vector operations. Shuffle operations are special because they execute cross-lane data movement and are important to build reduction functions, for example. Load and gather operations copy $\mathcal{W}_\mathtt{T}$ values from consecutive or indexed—respectively—memory locations. The reverse operations, store and scatter, also exist. The remaining depicted operations show the concurrent execution of $\mathcal{W}_\mathtt{T}$ operations on $\mathcal{W}_\mathtt{T}$ different operands.

1.3 SIMD OPERATIONS

The set of possible SIMD operations closely mirrors the available scalar instructions. Accordingly, there are instructions for arithmetic, compare, and logical operations. Figure 1.2 gives an overview for $\mathcal{W}_\mathtt{T} = 4$. However, SIMD instruction sets typically are not complete. For instance, the SSE 32-bit integer multiply instruction did not exist until the introduction of SSE4. Also division (especially for integers) is a likely candidate for omission in SIMD instruction sets (cf. [41, 70]). This difference in available operations is one reason why it is not easy to predict the speedup from vectorization of a given algorithm.

1.3.1 ALIGNMENT

SIMD programming requires special attention to alignment, otherwise the program may perform worse or even crash. The C++ standard defines alignment in [48, §3.11]:

> Object types have alignment requirements which place restrictions on the addresses at which an object of that type may be allocated. An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated.

For fundamental types the alignment requirement normally equals the size of the type.

A vector register requires $\mathcal{W}_{\mathtt{T}} \cdot \mathcal{S}_{\mathtt{T}}$ Bytes when stored in memory. SIMD architectures typically require the natural alignment of vectors in memory to be equal to their sizes. Thus, the alignment requirement is $\mathcal{W}_{\mathtt{T}}$ times higher than for type **`T`** ($\mathit{alignof}(\mathtt{VT}) = \mathcal{W}_{\mathtt{T}} \cdot \mathit{alignof}(\mathtt{T})$, where **`VT`** is the type for a vector of **`T`**). Only every $\mathcal{W}_{\mathtt{T}}{}^{th}$ address in an array of **`T`** is a valid aligned address for vector register load and store operations. One may therefore consider the memory to be an interleaving of memory that is local to each SIMD lane. Consequently, memory is partitioned into $\mathcal{W}_{\mathtt{T}}$ parts for a given type **`T`**.

Nevertheless, there are several mechanisms to break this natural partitioning of the memory and assign data to different SIMD lanes. The most obvious of them are unaligned loads and stores. These are usable with an arbitrary offset into an array. It is also possible to execute two aligned loads and stores together with one or more vector shuffle instructions. Additionally, for greatest flexibility, a SIMD instruction set might support gather and scatter instructions. These take a base address and a vector of offsets to load or store the entries of a vector from/to arbitrary memory locations.

1.3.2 COMPARISONS

Another important set of instructions are comparisons. In contrast to compare operations of fundamental types, there cannot be a single `true` or `false` answer for a SIMD comparison. Instead there must be $\mathcal{W}_{\mathtt{T}}$ boolean values in the result. This implies that such an instruction cannot sensibly set any of the flags that most architectures use for branching instructions.

A vector of $\mathcal{W}_{\mathtt{T}}$ boolean values is called a *mask*. There exist different implementations for such masks. They can either use the same registers as the data vectors or they are stored in specialized mask registers. Special mask registers can thus be optimized to store only $\max\{\mathcal{W}_{\mathtt{T}} \forall \mathtt{T}\}$ Bits. If the mask is stored in a vector register, it is useful to have an instruction converting the mask to an integer where every bit corresponds to one boolean value of the mask.

1.4 CURRENT SIMD INSTRUCTION SETS

There have been many different SIMD computers, but the most relevant SIMD instruction sets at the time of writing are the ones based on x86, ARM, and PowerPC architectures.

1.4.1 POWERPC

The PowerPC architecture is most relevant for its use in the Xbox 360 and PlayStation 3 game consoles and the Blue Gene supercomputers. These CPUs provide SIMD instructions known as AltiVec, VMX, VMX128, or Velocity Engine. *PowerPC Microprocessor Family : Vector / SIMD Multimedia Extension Technology Programming Environments Manual* [70] documents the details: The vector registers are 128 bits wide and can store sixteen 8-bit, eight 16-bit, or four 32-bit integers or four 32-bit single-precision floats. Thus, 64-bit data types, especially double-precision floats, are not supported in AltiVec vector registers. In addition to typical arithmetic operations the instruction set contains FMA[11] operations. Also, AltiVec contains a generic permute instruction for very flexible data reorganization from two source registers.

1.4.2 ARM

The ARM architecture is becoming more and more relevant, not only because of its use in mobile products, but moreover because of its potential as energy-efficient general-purpose computing platform. Most current ARM designs include a SIMD instruction set called NEON, which was introduced with the ARM Cortex™-A8 processor (ARMv7 architecture). *Introducing NEON™* [47] provides an introduction. The vector registers can be accessed as sixteen 128-bit or thirty-two 64-bit registers. There is no need to explicitly switch between 128-bit and 64-bit vector registers as the instruction used determines the view on the register bank. If the VFPv3 instructions are implemented, the vector registers additionally alias the floating-point registers. The registers can pack 8-bit, 16-bit, 32-bit, or 64-bit integers or 32-bit single-precision floats. Optionally, NEON supports fast conversion between single-precision floats and fixed-point, 32-bit integer, or half-precision floats.

Even without the NEON extension, ARM CPUs (since ARMv6) support a small set of SIMD instructions on integer values packed into standard 32-bit registers.

11 Fused Multiply-Add

1.4.3 x86

Intel introduced the first SIMD extension to the x86 family in 1997 with the "Pentium processor with MMX Technology" [69]. This extension is still supported but for all practical purposes superseded by the SSE and SSE2 extensions that followed in 1999 and 2001. SSE provides eight (sixteen in 64-bit mode) vector registers which are 128 bits wide. It supports 8-bit, 16-bit, 32-bit, and 64-bit integers in addition to 32-bit single-precision and 64-bit double-precision floats. The available instructions for the different integer types differ slightly and some useful integer instructions were only added with later SSE revisions. For all recent microarchitectures the usage of SIMD instructions improves computational throughput by a of factor $\mathcal{W}_\mathtt{T}$.[12] With the Intel SandyBridge and AMD Bulldozer microarchitectures (both released in 2011) AVX was introduced, doubling the SIMD register width on x86. However, AVX only shipped full-width arithmetic instructions for single-precision and double-precision floats. Thus, $\mathcal{W}_\mathtt{T}$ effectively only doubled for `float` and `double`. With the Intel Haswell microarchitecture (released in 2013) AVX2 was introduced. AVX2 includes the missing integer arithmetic instructions. Additionally, AVX2 includes gather instructions able to optimize indirect array reads. AMD introduced FMA instructions with the Bulldozer microarchitecture in 2011. The use of FMA instructions is required on Bulldozer for full performance. Intel introduced FMA with Haswell and thus doubled SIMD performance relative to the previous microarchitecture, as well.

1.4.4 INTEL XEON PHI

The Intel Xeon Phi is a highly parallel, high performance computing accelerator to compete with GPUs[13] for general-purpose computing. It is built from many small, relatively slow, but 16-way vectorized CPU cores. The Xeon Phi provides 32 vector registers of 512 bits width. It supports sixteen 32-bit integers, sixteen 32-bit single-precision floats, or eight 64-bit double-precision floats. There is limited support for 64-bit integers. Other data types are supported via cheap conversions at load or store.

On the Xeon Phi, one vector register is exactly as large as one cache line. Consequently, any unaligned vector load or store is certain to be more expensive than an aligned access. This is clearly visible in the Xeon Phi's instruction set where unaligned loads and stores each require two instructions: one instruction per cache line. Furthermore, the large register size has implications on locality considera-

12 This is not entirely true for the AMD Bulldozer microarchitecture. The effective SIMD width is that of SSE. AVX support is implemented as execution of two 128-bit operations.

13 Graphics Processing Units

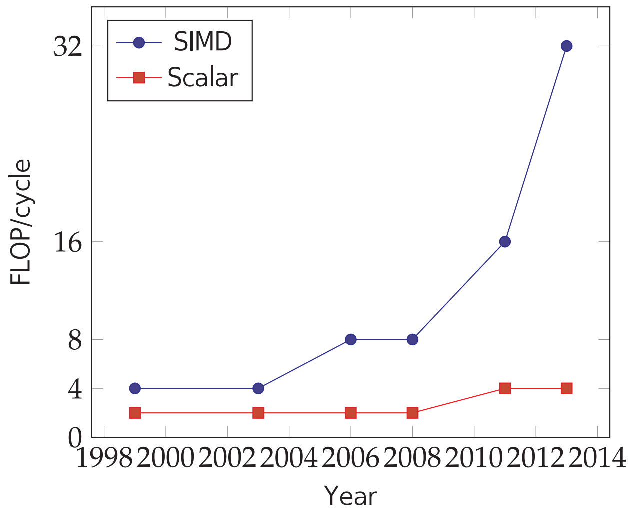


Figure 1.3: The maximum single-precision FLOP/cycle when using SIMD/scalar instructions on a single thread on x86(-64) (maximum of AMD and Intel CPUs) plotted against year of releases of new microarchitectures. [74, 26, 41]

tions since loading one full vector does not fetch any other related or unrelated data.

1.5 THE FUTURE OF SIMD

In the last years we have seen a steady rise in importance and performance relative to scalar instructions for SIMD (see Figure 1.3). This development is unlikely to reverse. It is consequential to expect that vector registers are going to become wider and/or that more efficient SIMD instructions will be released. The AVX instruction coding (VEX prefix) has a 2-bit field encoding the width of the register. Values 0 and 1 are used for 128-bit and 256-bit width, respectively. Values 2 and 3 are likely to be used for 512-bit and 1024-bit wide vectors.

GPUs have shown the great potential of SIMD in general purpose computing. The important GPU architectures are internally similar to SIMD architectures, even though they are typically able to make independent progress on all the “quasi-threads”, requiring an explicit barrier for inter-lane communication. Their programming models are becoming more flexible with every release and will likely converge more and more to general purpose programming models (such as full C++ support).

1.6 INTERACTION WITH OTHER FORMS OF PARALLELISM

Hardware designers have implemented several dimensions of parallel execution in current CPU architectures to achieve high performance. It is therefore certainly possible that, if use of the SIMD dimension is increased, a different dimension has less opportunity to parallelize the execution. This section will discuss the issue.

1.6.1 INSTRUCTION LEVEL PARALLELISM

CPUs use pipelining to start execution of subsequent instructions in the instruction stream while previous ones have not been retired yet. However, in general an instruction cannot start before all its input registers are ready, implying that all instructions writing to those registers need to have retired. Therefore, a stream of instructions where each one depends on the output of the previous one cannot benefit from pipelining. It is often possible to execute several serial instruction streams on independent data streams in the same time required to process a single data stream. Consequently, compilers (and sometimes also developers) apply loop unrolling to create this data-parallel stream of independent calculations.

Some transformations from a scalar implementation to a SIMD implementation may reduce the opportunity for pipelined execution. This is especially the case with "vertical vectorization" (cf. [59]), where several independent, serially executed instructions are transformed into a single SIMD-parallel instruction. In such a case the execution units are stalling for longer durations in the SIMD implementation. It is one of the reasons why vectorization sometimes does not yield the expected speedup. Especially, because current compilers are sometimes not as smart about loop unrolling SIMD code as they are for scalar code.[14] Manual loop unrolling of SIMD code often improves the vectorization results.

Superscalar execution is another dimension of parallelism present in current CPUs [40]. The CPU schedules instructions to different execution units, which work independently. Typically, the execution units are specialized for a specific set of instructions. For example, the Intel SandyBridge CPU can schedule execution of up to 6 micro-ops every cycle. Mapped to actual instructions this could be one floating-point `add/sub`, one floating-point `mul`, and two 128-bit load instructions [41]. Superscalar execution therefore does not compete with SIMD parallelization. It equally accelerates SIMD and scalar instructions.

Speculative execution is an optimization tied to branch prediction. The CPU predicts which path the code will take at a given branch instruction. It will then continue to execute the predicted code path (only operations that can be rolled back) as if the branch was taken. On a mis-prediction this work will be rolled back and the other branch will be executed. SIMD code often has fewer branches because branch

14 This is not a fundamental problem, but just missing optimizations in the compiler.

conditions are tied to scalar values and not the whole vector. Consequently, SIMD code has to execute all branches and use conditional assignment to merge the results. SIMD code therefore does not benefit as much from speculative execution and might not yield a good speedup for algorithms with many conditionals.

1.6.2 MULTIPLE CORES

Common frameworks for thread-parallelization, such as OpenMP [68] and TBB[15] [45] provide easy to use multi-threaded execution of data-parallel algorithms (and more) via fork-join execution. The common parallel-for construct splits the work of a given loop into several chunks that are executed on separate worker threads. This parallelization strategy is very efficient because it makes the code scale from low to high CPU core counts rather easily. However, SIMD relies on data-parallelism as well. Therefore, on the one hand, vectorization and multi-threading may compete for the same inherent parallelism. On the other hand, the combination of multi-threading and vectorization may also complement each other, because in contrast to SIMD execution threads allow concurrent execution of different/diverging tasks.

1.7 COMMON OPTIMIZATION CHALLENGES

If a given scalar code is transformed into SIMD code the throughput of arithmetic (and logic and compare) operations can increase as much as by a factor of $\mathcal{W}_{\mathrm{T}}$. For almost every algorithm this requires an equal increase in load/store throughput. Thus, the size of the working set increases and the cache may be under considerably higher pressure.

The load/store unit may also be a bottleneck. For example, the Intel SandyBridge and AMD Bulldozer microarchitectures both can execute up to two 128-bit loads and one 128-bit store per cycle [74, 41]. Thus, only one AVX vector can be read from L1 cache per cycle, and only one AVX vector can be stored to L1 cache every second cycle. This is a considerable limitation, given that these CPUs can execute two AVX floating-point instructions or even three AVX logical operations per cycle. Intel therefore improved the Haswell microarchitecture to allow two 256-bit loads and one 256-bit store per cycle [40].

Figure 1.4 shows the result of a benchmark (cf. Appendix A) testing the floating-point operations throughput for different scenarios. It turns out that SIMD improves the throughput for every tested case. Even if very few operations are executed over a very large working set, the result is slightly improved by using SIMD. This is visible in the lower plot showing the speedup factor $\frac{\text{AVX}}{\text{Scalar}}$.

15 Intel Threading Building Blocks

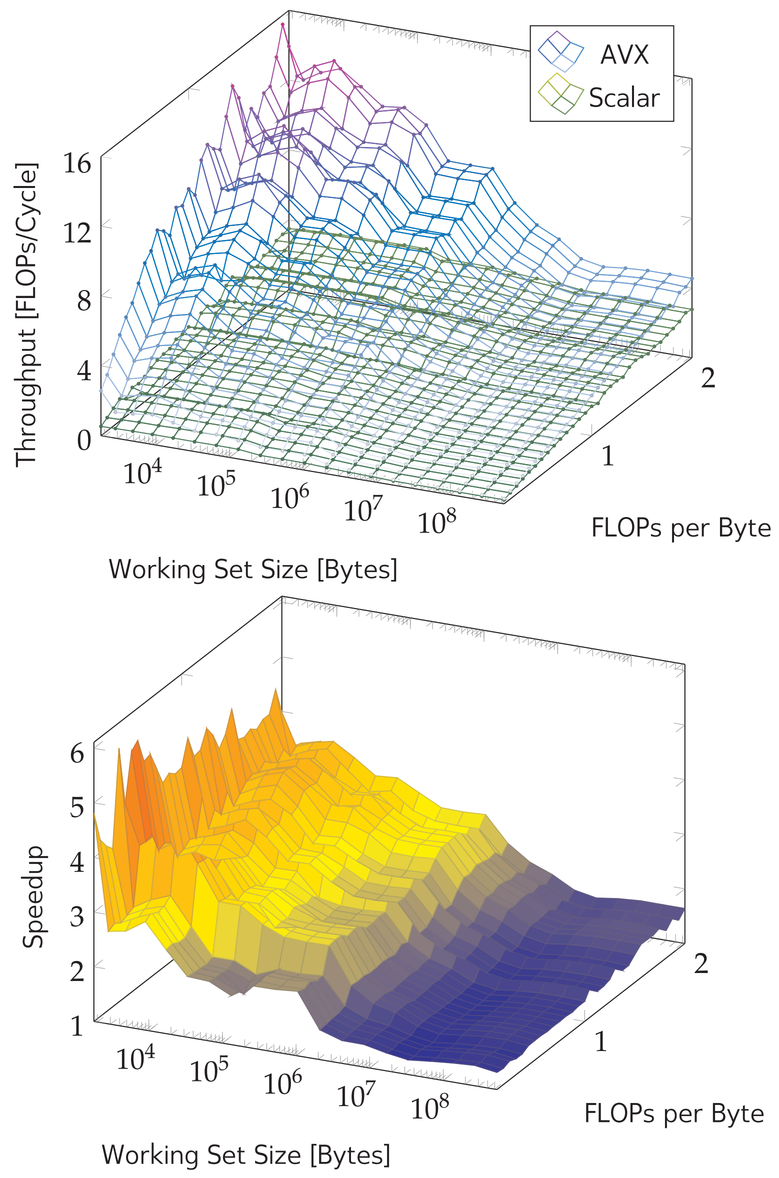


Figure 1.4: The plot shows the measured (single-precision) FLOPs per clock cycle depending on the working set size and the amount of calculation per data access (loads and stores combined). The benchmark was compiled and executed once with AVX instructions and once without SIMD instructions for an AMD Bulldozer CPU. It uses FMA instructions in both cases. However, the number of values that are processed in parallel depend on $\mathcal{W}_{\texttt{float}}$.
The speedup plot shows the quotient of the AVX throughput relative to the scalar throughput ($\frac{\text{AVX}}{\text{Scalar}}$). (Note that for AVX, with $\mathcal{W}_{\texttt{float}} = 8$, the best theoretical speedup is 800%. See Appendix A for the benchmark code.)

1.8 CONCLUSION

Software developers and designers have to consider SIMD. Developers need to understand what SIMD can do and how it can be used. Only then a developer can design data structures and algorithms which fully utilize modern CPUs. Especially compute intensive programs need to use the SIMD capabilities, otherwise CPU utilization will only be at 6–25% of theoretical peak performance. However, programs that are bound by memory accesses can also benefit from vectorization since SIMD can make many memory operations more efficient (cf. Figure 1.4). Only tasks such as user interaction logic (and other inherently sequential codes) often have no use for SIMD.

From the discussion of different forms of parallelism we can conclude that it is helpful for developers to understand how SIMD and other forms of parallelism work. Blindly vectorizing a given code might not yield the expected performance improvement. Therefore, the interface for data-parallel programming should provide as much guidance as possible. Such an interface needs to be designed from use-cases of important parallelization patterns, and not only from existing hardware capabilities. This design process will additionally lead to an abstraction that can hide the substantial differences in hardware.

2 LOOP-VECTORIZATION

It is a common pattern of software development to execute a sequence of operations on different input values. Programming languages support this pattern with loop statements. In particular, the `for`-loop is designed for iteration over an index or iterator range. In many cases the individual iterations of the loop are independent from one another (or could be expressed in such a way) and could therefore be executed in any order and even in parallel. This observation has led to many parallelization interfaces that build upon parallelization of loop iterations.

Loop-vectorization, including auto-vectorization, is likely the most prevalent method of vectorization. Since this work presents a solution for SIMD programming in C++ this chapter will present the main alternatives and discuss their limitations and issues. The following chapters on Vc (Part II) will show that these issues are solved / do not occur with SIMD types.

2.1 DEFINITIONS

COUNTABLE LOOPS A common requirement of tool-based vectorizers is that the number of iterations a loop will execute can be determined before the loop is entered. Loops where this condition holds are called *countable*.

VECTORIZATION The process of transforming a serial/scalar code into vector/SIMD code is called *vectorization*. This transformation is relatively simple to apply for (inner) loops. As a simple example consider Listing 2.1. (The iteration-count can be determined before the loop is entered, making the loop *countable*.) The iterations are all independent, because `&d[i] != &d[j]` $\forall i \neq j$. Thus, the loop may be transformed into $\mathtt{N}/\mathcal{W}_{\mathtt{float}}$ iterations where each remaining iteration executes

```
void func(float *d) {
  for (int i = 0; i < N; ++i) {
    d[i] += 1.f;
  }
}
```

Listing 2.1: A simple countable loop of independent calculations.

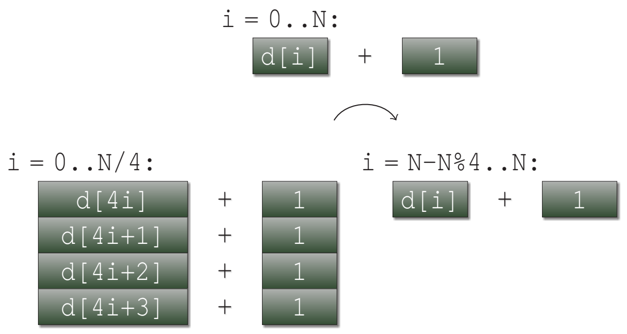


Figure 2.1: Illustration of the loop transformation which the vectorizer applies: A loop over N additions is transformed into a vectorized loop and possibly a scalar prologue and/or epilogue loops. This transformation is the basic strategy for (automatic or manual) vectorization of most data-parallel problems.

$\mathcal{W}_{\texttt{float}}$ additions with one instruction (Figure 2.1). Plus a remainder loop in case N is not a multiple of $\mathcal{W}_{\texttt{float}}$.

AUTO-VECTORIZATION Auto-vectorization describes the process of automatically locating vectorizable code and applying the necessary code transformations. In most cases this is done in an optimization step of the compiler. Alternatively, there are tools (i.e. meta-compilers) translating scalar code to (compiler-specific) SIMD code. The auto-vectorizer identifies vectorizable loops or possibly vectorizable sequences of scalar operations on adjacent values in an array.

EXPLICIT LOOP-VECTORIZATION For a given loop, the developer can annotate the loop as SIMD loop. There are two basic approaches to these loop annotations:

1. The loop annotation tells the compiler that the developer expects the loop to be vectorizable under the transformation rules of the auto-vectorizer. Therefore, the developer can reason about his/her code with serial semantics[1]. On the other hand, if auto-vectorization fails it is now possible for the compiler to generate more diagnostic information.[2] In addition to stating the intent, the annotation typically supports additional clauses able to guide the vectorizer to better optimizations.

2. The loop annotation alters the semantics of the loop body. Serial semantics do not apply anymore. Instead vector semantics are used and the compiler does not have to prove that serial semantics and vector semantics produce

1 "Serial semantics" characterizes the normal C++ *sequenced before* rule [48, §1.9 p13].
2 At least with GCC[3] 4.9.1 no useful diagnostics were printed when the loop failed to be vectorized (even with the `-Wopenmp-simd` flag).

the same result. Thus, cross-iteration data-dependencies (e.g. `for (int i = 0; i < N - 1; ++i) a[i] += a[i + 1];`) will not inhibit vectorization, enabling a new class of algorithms to be vectorizable via loops. However, it can lead to subtle bugs where the developer (erroneously) reasons about the code with serial semantics.

The following discussion will only refer to the second variant of explicit loop-vectorization, because it is the more general and powerful solution. The first variant only slightly improves over the auto-vectorization status-quo.

VECTOR SEMANTICS A thorough definition of vector semantics is an important prerequisite for an implementation of explicit loop-vectorization in a programming language. The Parallelism TS [49] states the following on the semantics of code executed with a vector-parallel execution policy:

> The invocations of element access functions [...] are permitted to execute in an unordered fashion [...] and unsequenced with respect to one another within each thread.

For a loop, this means that individual iterations can make independent progress, progress in lock-step, or in any conceivable interleaving. Riegel [71] defines a possible forward progress guarantee for "weakly parallel execution":

> The implementation does not need to ensure that a weakly parallel EA[4] is allowed to execute steps independently of which steps other EAs might or might not execute.

Geva et al. [30] state a formal definition of the wavefront execution model (cf. [35]), which is a stricter progress guarantee for vector semantics:

> A SIMD loop has logical iterations numbered 0, 1, ..., N-1 where N is the number of loop iterations, and the logical numbering denotes the sequence in which the iterations would execute in the serialization of the SIMD loop. The order of evaluation of the expressions in a SIMD loop is a partial order, and a relaxation of the order specified for sequential loops. Let X_i denote evaluation of an expression X in the i^{th} logical iteration of the loop. The partial order is:
>
> For all expressions X and Y evaluated as part of a SIMD loop, if X is sequenced before Y in a single iteration of the serialization of the loop and $i \leq j$, then X_i is sequenced before Y_j in the SIMD loop.
>
> [*Note:* In each iteration of a SIMD loop, the "sequenced before" relationships are exactly as in the corresponding serial loop. — *end note*]

4 EA (Execution Agent) is defined by *ISO/IEC 14882:2011* [48, §30.2.5.1] as: "An execution agent is an entity such as a thread that may perform work in parallel with other execution agents."

This overview shows that there is no consensus—at this time—for the specifics of vector semantics. Any further discussion in this document involving vector semantics applies to all variants of vector semantics, though.

2.2 AUTO-VECTORIZATION

2.2.1 USES FOR AUTO-VECTORIZATION

It is the job of a compiler to create the best machine code from a given source. The executable code should make use of the target's features (in terms of instruction set and micro-architecture) as much as is allowed under the "as-if" rule of C++ [48, §1.9 p1]. Therefore, auto-vectorization is an important optimization step. Many codes can benefit without any/much extra effort from the developer's side. This is important for low-budget development or developers without experience in parallel programming. It can also be used for highly optimized codes. Though, in this case explicit loop-vectorization should be preferred, as the implicit expression of parallelism is fragile in terms of maintainability.

2.2.2 INHIBITORS TO AUTO-VECTORIZATION

Intel documented the capabilities and limitations of the ICC[5] auto-vectorizer in detail [4]. They point out several obstacles and inhibitors to vectorization:

NON-CONTIGUOUS MEMORY ACCESSES: Non-contiguous memory accesses lead to inefficient vector loads/stores. Thus the compiler has to decide whether the computational speedup outweighs the inefficient loads/stores.

DATA DEPENDENCIES: Data dependencies between the iteration steps make it hard or impossible for the compiler to execute loop steps in parallel.

COUNTABLE: If the loop is not countable the compiler will not vectorize.

LIMITED BRANCHING: Branching inside a loop leads to masked assignment. Instead of actual branches in the machine code, the auto-vectorizer has to emit code that executes all branches and blend their results according to the condition on each SIMD lane. If the compiler determines that the amount of branching negates the improvement of vectorization it will skip the loop.

OUTER LOOPS: Outer loops (loops containing other loops) are only vectorized after certain code transformations were successful.

FUNCTION CALLS: Function calls inhibit vectorization. However, functions that can be inlined and intrinsic math functions are exceptions to this rule.

5 Intel C++ Compiler

THREAD INTERACTION: Any calls to mutexes, or atomics inhibit auto-vectorization.

Auto-vectorization in GCC [5] additionally documents that for some SIMD transformations the order of arithmetic operations must be modified. Since this kind of optimization can change the result when applied to floating-point variables it would deviate from the C++ standard. The standard specifies that operators of equal precedence are evaluated from left to right. Consequently, *Auto-vectorization in GCC* [5] recommends the `-ffast-math` or `-fassociative-math` flags for best auto-vectorization results.

2.2.3 LIMITATIONS OF AUTO-VECTORIZATION

In most cases, auto-vectorization cannot instantly yield the best vectorization of a given algorithm (if at all). This is in part due to the limited freedom the compiler has in transforming the code, such as aliasing issues, fixed data structures, pointer alignment, function signatures, and conditional statements. Another important part is that the user did not express the inherent data-parallelism explicitly. The compiler has to infer information about the problem that got lost in translation to source code. Finally, the user has to limit himself to a subset of the language as listed in Section 2.2.2.

If a user wants to vectorize his/her code via auto-vectorization, it is necessary to let the compiler report on its auto-vectorization progress and use this information to adjust the code for the needs of the vectorizer. Sometimes this can require larger structural changes to the code, especially because data storage must be transformed from arrays of structures to structures of arrays. Additionally, users should add annotations about the alignment of pointers to improve vectorization results.

2.2.3.1 ALIASING

Auto-vectorization relies on iterations over arrays to express the per-iteration input and output data. Since structures of arrays work best for the vectorizer, the loop body typically dereferences several pointers to the same fundamental data type. The compiler must account for the possibility that these pointers are equal or point to overlapping arrays. Then, any assignment to such an array potentially alters input values and might create cross-iteration dependencies.

```
float add_one(float in) {
  return in + 1.f;
}
```

Listing 2.2: A simple function.

```
float *data = ...
for (int i = 0; i < N; ++i) {
  data[i] = add_one(data[i]);
}
```

Listing 2.3: A function call in a loop.

2.2.3.2 FIXED DATA STRUCTURES

A major factor for the efficiency of a vectorized algorithm is how the conversion from $\mathcal{W}_T$ scalar objects in memory to a single vector register and back is done. The compiler has no freedom to improve the data layout, which is fully defined by the types used by the algorithm. At the same time, the scalar expression of the algorithm conceals the problem from the developer who often is oblivious to the limitations of the vector load and store operations of the hardware.

2.2.3.3 POINTER ALIGNMENT

In most cases the compiler cannot deduce whether a given pointer uses the necessary over-alignment for more efficient vector load and store instructions. Without extra effort, and an understanding of hardware details, the user will therefore create a more complex vectorization of the algorithm than necessary.

2.2.3.4 FUNCTION SIGNATURES

Consider a simple function like Listing 2.2. A function defines an interface for data in- and/or output. There are fixed rules how to translate such code to a given ABI[6]. For example the `in` parameter has to be stored in the bits 0–31 of register `xmm0` on x86_64 Linux. Unless the compiler is allowed/able to inline the function, the function itself cannot be vectorized. Neither can a calling loop, such as Listing 2.3, be vectorized in this case.

In theory, the above limitation can be solved with LTO[7]. With LTO the compiler has more opportunity to inline functions since it still has access to the abstract syntax tree of the callee when the optimizer vectorizes the caller loop.

6 Application Binary Interface
7 Link Time Optimization

2.2.3.5 CONDITIONAL STATEMENTS

Many algorithms require conditional execution which depends on the actual values of the input data. Since the algorithm is expressed with scalar types, the user has to use `if`-statements (or `while`, `for`, or `switch`). These cannot be vectorized as actual branches. Instead, the vectorized code has to execute all cases and implicitly mask off assignments according to the conditions in the different vector lanes. This leads to more code that needs to be executed and the user may be completely unaware of the cost of conditionals (and possible alternatives), unless (s)he has learned how the vectorizer and the target hardware works.

2.3 EXPLICIT LOOP-VECTORIZATION

ICC has integrated explicit loop-vectorization (SIMD-loops) in their compiler via the `#pragma simd` extension [42]. Intel considers this extension important enough that they are pursuing inclusion of this feature in the C++ standard (cf. [30, 35]). The extension is a significant improvement over auto-vectorization, where the developer has no mechanism to ensure SIMD parallel execution of a given loop.

All the limitations described in Section 2.2.3 still apply to explicit loop-vectorization. In contrast to auto-vectorization, the user explicitly expresses that the algorithm is data-parallel, which improves diagnostics and maintainability. In addition, optional clauses to the SIMD-loop syntax allow the user to guide the compiler to a more efficient vectorization.

Vector semantics (as defined in Section 2.1) imply the following restrictions, which are correct and well-defined scenarios with serial semantics:

- The behavior is undefined for loops with cross-iteration dependencies, unless correctly annotated with a clause such as `safelen`. It would be desirable to require compilers to make SIMD-loops with such dependencies ill-formed. This is impossible, though, because the dependencies might be hidden behind pointer aliasing. The issue is thus only detectable at run time.

- Thread synchronizing operations, such as mutexes, may deadlock. Current implementations of explicit loop-vectorization either have to declare synchronizing operations either as undefined behavior or ill-formed.

- If an exception is thrown in a vector loop, the behavior is undefined or may result in a call to `std::terminate`.[8]

If such a case occurs in a (non-inline) function called from the SIMD-loop, the compiler cannot recognize the condition, in which case auto-vectorization would be inhibited. With vector semantics it is well-defined behavior to call the function

8 `std::terminate` is the consequence of throwing a second exception during stack unwinding [48].

for each entry in the SIMD vector in sequence, though. This can lead to deadlock, surprising side-effects (e.g. global variables), or inconsistent exceptions with a possible call to `std::terminate`.

The user must therefore be aware of the different semantics and the functions (s)he calls. Especially calls into library functions may be fragile as a new release of the library might become unsafe to use in a SIMD-loop.

3

INTRODUCTION TO SIMD TYPES

Perhaps the greatest strength of an object-oriented approach to development is that it offers a mechanism that captures a model of the real world.

— Grady Booch (1986)

3.1 THE SIMD VECTOR TYPE IDEA

Most of the current commodity hardware supports SIMD registers and instructions. However, C++ projects that target these systems have no standardized way to explicitly use this part of the hardware. The solutions that currently serve as best practice are auto-vectorization (i. e. pure scalar code / cf. Section 2.2), OpenMP 4 [68] and Cilk Plus [44] SIMD loops (cf. Section 2.3), vendor-specific SIMD intrinsics (cf. Section 3.2), and (inline) assembly. C++ itself only provides types and operators for scalar integer and floating-point registers and instructions. The standard leaves most of the mapping of type names to exact register widths implementation-defined. SIMD registers and operations are simply ignored in the standard and left to "as-if" transformations of the optimizer (auto-vectorization).

If C++ were to provide types that map to implementation-defined SIMD registers and instructions then SIMD instructions would be just as accessible as scalar instructions (see Figure 3.1). Such a SIMD type should, in most respects, behave analogous to the scalar type, in order to make SIMD programming intuitively usable for any C++ developer. The obvious difference to scalar types is that instead of storing and manipulating a single value, it can hold and manipulate $\mathcal{W}_\mathtt{T}$ values in parallel.

Since the differences between different target microarchitectures are a lot more significant in SIMD instructions than in scalar instructions, the portability concerns with these types are a lot more apparent than with scalar types. Therefore, while the type needs to be target-dependent, the interface must be target-agnostic. The type thus needs to express data-parallel execution in an abstract form. It should not be designed specifically for SIMD execution, which is just one form of efficient

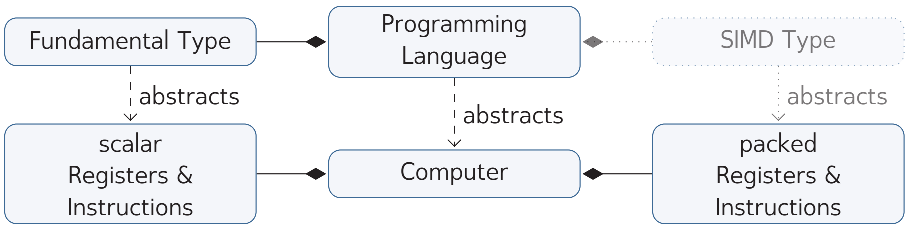


Figure 3.1: Programming languages abstract how computers work. The central processing units of the computer is controlled via registers and instructions. Most languages, notably C++, use types and their associated operators to abstract registers and instructions. Though, since the advent of SIMD registers and instructions there exists a gap in the abstraction.

data-parallel execution. Such a strong design focus on SIMD could make the type less portable, thus less general, and therefore less useful.

On the other hand, a type that cannot translate to the most efficient SIMD instructions (in all but maybe a few corner cases) would not be useful either. Users that want to explicitly optimize for a given target architecture should be able to do so with the SIMD type. The design goal is therefore to make the vector types (together with a properly optimizing compiler) as efficient as hand-written assembler SIMD code.

3.2 SIMD INTRINSICS

All major C++ implementations provide SIMD intrinsics as an extension to the standard. These extensions provide types and functions that map directly to the available SIMD registers and instructions. One of the most widespread extensions are x86 SIMD intrinsics for MMX[1], SSE, AVX.

Another important extension are builtin vector types, which provide more contextual information to the middle-end and back-end of the compiler. The GCC and clang implementations of vector builtins support operators for the vector types and retain more information for the optimizer about the per-entry operations that are executed. This allows the compiler to be smarter about optimizing sequences of SIMD operations.

1 Multimedia Extensions

3.3 BENEFITS OF A SIMD VECTOR TYPE

A SIMD vector type constitutes the low-level interface to SIMD programming.[2] It is possible to build higher levels of abstraction using standard C++ facilities on top of these types. If this lowest level of SIMD programming is not provided then users of C++ will be constrained to work within the limits of the provided higher level abstraction. Even if such an abstraction were very efficient for most problems, it could lead to a situation where an additional and unrelated means of SIMD programming (outside of standard C++) were still required.

In some cases the compiler might generate better code if only the intent is stated instead of an exact sequence of operations. Therefore, higher-level abstractions might seem preferable to low-level SIMD types. In my experience this is not the case because programming with SIMD types makes intent very clear and compilers can optimize sequences of SIMD operations. This is effectively identical to optimizations on scalar operations.

SIMD types themselves do not lead to an easy and obvious answer to efficient and easily usable data structures. However, SIMD types reveal inefficient data structures, as they become hard or awkward to use. This can guide developers to create more efficient data structures for vectorization. Chapter 10 shows a high-level interface to data structure vectorization, which can make creation of proper vectorizable data structures easier or even completely transparent for a user of a given interface (such as shown in Chapter 11). On the other hand, a SIMD programming abstraction that completely hides vectorized memory accesses behind scalar memory accesses requires compiler diagnostics to guide the developer to more efficient data structures.

One major benefit from SIMD types is that the programmer can gain an intuition for SIMD. This subsequently influences further design of data structures and algorithms to better suit SIMD architectures.

Additionally, there are already many users of SIMD intrinsics (and thus a primitive and non-portable form of SIMD types). Providing a cleaner and portable SIMD API[3] would provide many of them with a better alternative. Thus, SIMD types in C++ would capture existing practice.

2 One could argue that target-specific intrinsics or builtins should be considered as the low-level interface. I believe this is the wrong approach, since the programming language should only provide abstractions for all the machines it may be compiled for and not concrete, target-specific interfaces.

3 Application Programming Interface

3.4 PREVIOUS WORK ON SIMD TYPES

3.4.1 INTEL C++ CLASS INTERFACE TO SSE

Intel has provided simple C++ wrapper classes with the Intel C++ Compiler. These classes use operator overloading to make intrinsics more convenient to use from C++. However, these classes have stopped short from becoming a more generic solution for data-parallel programming. The types are directly tied to a specific SIMD register width. Therefore, these types can only be used for compatible targets. Code written with these types does not scale to newer hardware, even if it is a compatible target. Access to scalar entries in a SIMD vector is provided, but breaks aliasing requirements, leading to subtle bugs.[4]

3.4.2 E.V.E.

The E.V.E. [21] library wraps the AltiVec instruction set of PowerPC CPUs. As the paper notes, it would be feasible to implement different target architectures, such as MMX/SSE2 and thus provide a portable API. The main emphasis was put on vectorization of calculations on larger data structures with more or less homogeneous treatment. The E.V.E. library built a basis for the NT^2 project [20], which now contains a more modern and more flexible SIMD abstraction called Boost.SIMD (Section 3.5.1).

3.4.3 MACSTL

Another library wrapping AltiVec exists, where MMX and SSE up to SSE3 were added later on, called macstl [62]. This library focuses on a main class for processing which is an improved STL[5] `valarray` implementation with vectorized operations on it. However, the flexibility for heterogeneous processing, required for the track finder, was missing. The library is not licensed as Free Software, and was thus unsuitable for the tracker.

3.4.4 KISEL/GORBUNOV'S HEADERS

The Kalman filter, the most compute-intensive part of the track finder, has previously been vectorized, which was presented in [33]. For this work, a small abstraction around SSE was developed. This work was evaluated and incorporated into the library presented here. In the mean time, this Kalman filter benchmark has been modified to use Vc, showing a slight improvement in performance [53].

4 This is normally not a problem for projects compiled with ICC, since the compiler per default does not enforce strict-aliasing rules.
5 Standard Template Library

3.5 OTHER WORK ON SIMD TYPES

Since its first public release of the Vc library in 2009 there have been several new projects to abstract SIMD types very similar to the basic idea discussed here. The main difference in the interface abstraction of Vc to the other libraries is that Vc abstracts data-parallel programming as independent from $\mathcal{W}_\mathtt{T}$ as possible. The other libraries encode $\mathcal{W}_\mathtt{T}$ in the type name or encode the target SIMD instruction set in the type name.[6]

3.5.1 BOOST.SIMD

The Boost.SIMD library[7] abstracts SIMD objects via the **`pack<T, N>`** class template [18, 17]. `N` may be omitted to compile for the native $\mathcal{W}_\mathtt{T}$ of the target. This makes **`pack<T>`** almost equal to **`Vc::Vector<T>`** except for the ABI incompatibility issue of **`pack<T>`** discussed in Chapter 6. The Vc mask type is available in Boost.SIMD as **`pack<logical<T>, N>`**, thus tying the mask API to the value vector API, which I specifically chose differently for the design of Vc (Section 5.2). Compare operators of **`pack<T>`** return **`bool`**, instead of the mask type for Boost.SIMD, requiring predicate functions to do vectorized compares instead. The implementation of write-masking is not as generic as in the Vc API: Boost.SIMD provides the `if_else` function that implements a vector blend and special functions like `selinc` and `seldec` to execute write-masked increment and decrement. The Vc API builds these expressions with standard C++ operators instead (Section 5.3).

In contrast to the Vc operators, Boost.SIMD uses expression templates with the operators for **`pack`**. This enables the library to do code transformations on longer expressions, such as fusing multiplications and additions/subtractions or reorder elementwise operations. Vc instead relies on the compiler optimizer to do these kind of optimizations, because expression templates can increase compile times significantly and make diagnostic output from ill-formed programs unreadable. Also from experience with Vc, a good optimizing compiler nowadays generates optimal machine code in almost all uses. The few remaining issues are solvable "missed-optimization" issues in the compiler. The important benefit of the Vc approach is that the vectorization quality does not depend on how many temporary values are captured in variables, which is a major "gotcha" with expression-template solutions.

6 Vc also encodes the SIMD instruction set in the type name, but it is an internal type name. The user-visible types are target-agnostic.

7 Boost.SIMD is not in boost yet. This is the intention of the developers, though.

3.5.2 OTHER LIBRARIES

The VCL Library [25] and the Generic SIMD Library [83] are two more implementations of C++ wrapper libraries around SIMD intrinsics. They are competing implementations for the ideas presented here and in earlier publications such as Kretz [55] and Kretz et al. [59].

Part II

Vc: A C++ Library for Explicit Vectorization

4

A DATA-PARALLEL TYPE

Programs must be written for people to read,
and only incidentally for machines to execute.
— Harold Abelson et al. (1996)

The SIMD vector class shall be an abstraction for the expression of data-parallel operations (cf. Section 3.1). If the target architecture of a compilation unit does not support SIMD instructions, but similar data-parallel execution, the expressed data-parallelism shall be translated accordingly. The following list states the desired properties for such a type:

- The value of an object of `Vector<T>` consists of $\mathcal{W}_{\mathtt{T}}$ scalar values of type `T`.
- The `sizeof` and `alignof` of `Vector<T>` objects is target-dependent.
- Scalar entries of a SIMD vector can be accessed via lvalue reference.
- The number of scalar entries ($\mathcal{W}_{\mathtt{T}}$) is accessible as a constant expression.
- Operators that can be applied to `T` can be applied to `Vector<T>` with the same semantics per entry of the vector. (With exceptions, if type conversions are involved. See below.)
- The result of each scalar value of an operation on `Vector<T>` does not depend on $\mathcal{W}_{\mathtt{T}}$.[1]
- The syntax and semantics of the fundamental arithmetic types translate directly to the `Vector<T>` types. There is an additional constraint for implicit type conversions, though: `Vector<T>` does not implicitly convert to `Vector<U>` if $\mathcal{W}_{\mathtt{T}} \neq \mathcal{W}_{\mathtt{U}}$ for any conceivable target system.

1 Obviously the number of scalar operations executed depends on $\mathcal{W}_{\mathtt{T}}$. However, the resulting value of each scalar operation that is part of the operation on `Vector<T>` is independent.

```
 1 namespace Vc {
 2   namespace target_dependent {
 3     template <typename T> class Vector {
 4       implementation_defined data;
 5
 6     public:
 7       typedef implementation_defined VectorType;
 8       typedef T EntryType;
 9       typedef implementation_defined EntryReference;
10       typedef Mask<T> MaskType;
11
12       static constexpr size_t MemoryAlignment = implementation_defined;
13       static constexpr size_t size() { return implementation_defined; }
14       static Vector IndexesFromZero();
15
16       // ... (see the following Listings)
17     };
18     template <typename T> constexpr size_t Vector<T>::MemoryAlignment;
19
20     typedef Vector<         float>  float_v;
21     typedef Vector<        double> double_v;
22     typedef Vector<  signed   int>    int_v;
23     typedef Vector<unsigned   int>   uint_v;
24     typedef Vector<  signed short>  short_v;
25     typedef Vector<unsigned short> ushort_v;
26   }  // namespace target_dependent
27 }  // namespace Vc
```

Listing 4.1: Template class definition for **Vector**<**T**>.
A concrete implementation for SSE2 could call the inner namespace `SSE`, use the intrinsic types **__m128**, **__m128d**, and **__m128i** for **VectorType**, a union for the `data` member, **T** & for **EntryReference**, $\frac{\mathcal{S}_{\texttt{VectorType}}}{\mathcal{S}_{\texttt{T}}}$ for `size()`, and $\mathcal{S}_{\texttt{VectorType}}$ for `MemoryAlignment`.

- The compiler is able to identify optimization opportunities and may apply constant propagation, dead code elimination, common subexpression elimination, and all other optimization passes that equally apply to scalar operations.[2]

4.1 THE VECTOR<T> CLASS TEMPLATE

The boilerplate of the SIMD vector class interface is shown in Listing 4.1. There are several places in this listing where the declaration says "*target-dependent*" or "*implementation-defined*". All of the following listings, which declare functions of the **Vector**<**T**> class (to insert on line 16), do not require any further implementation-specific differences. All these differences in **Vector**<**T**> are fully captured by the code shown in Listing 4.1.

2 In practice, there are still a few opportunities for compilers to improve optimization of SIMD operations.

The only data member of the vector class is of an implementation-defined type (line 4). This member therefore determines the size and alignment of `Vector<T>`. Therefore, the SIMD classes may not contain virtual functions. Otherwise, a virtual table were required and thus objects of this type would be considerably larger (larger by the minimum of the pointer size and the alignment of `VectorType`).

4.1.1 MEMBER TYPES

The member types of `Vector<T>` abstract possible differences between implementations and ease generic code for the SIMD vector types.

`VectorType`
: (line 7) is the internal type for implementing the vector class. This type could be an intrinsic or builtin type. The exact type that will be used here depends on the compiler and compiler flags, which determine the target instruction set. Additionally, if an intrinsic type is used it might not be used directly (on line 4) but indirectly via a wrapper class that implements compiler-specific methods to access scalar entries of the vector.

The `VectorType` type allows users to build target- and implementation-specific extensions on top of the predefined functionality. This requires a function that returns an lvalue reference to the internal data (line 4). See Section 4.10 for such functions.

`EntryType`
: (line 8) is always an alias for the template parameter `T`. It is the logical type of the scalar entries in the SIMD vector. The actual bit-representation in the SIMD vector register may be different to `EntryType`, as long as the observable behavior of the scalar entries in the object follows the same semantics.

`EntryReference`
: (line 9) is the type returned from the non-const subscript operator. This type should be an lvalue reference to one scalar entry of the SIMD vector. It is not required for `EntryReference` to be the same as `EntryType &`. Consider an implementation that uses 32-bit integer SIMD registers for `Vector<short>`, even though a `short` uses only 16 bits on the same target. Then `EntryReference` has to be an lvalue reference to `int`. If `EntryReference` were declared as `short &` then sign extension to the upper 16 bits would not work correctly on assignment.

`MaskType`
: (line 10) is the mask type that is analogous to `bool` for scalar types. The type is

used in functions that have masked overloads and as the return type of compare operators. A detailed discussion of the class for this type is presented in Chapter 5.

4.1.2 CONSTANTS

`SIZE()` The vector class provides a static member function (`size()`) which identifies the number of scalar entries in the SIMD vector (line 13). This value is determined by the target architecture and therefore known at compile time. By declaring the `size()` variable `constexpr`, the value is usable in contexts where constant expressions are required. This enables template specialization on the number of SIMD vector entries in user code. Also it enables the compiler to optimize generic code that depends on the SIMD vector size more effectively. The `size()` function additionally makes **`Vector<T>`** implement the standard container interface, and thus increases the reusability in generic code.[3]

`MemoryAlignment` The `MemoryAlignment` static data member defines the alignment requirement for a pointer passed to an aligned load or store function call of **`Vector<T>`**. The need for a `MemoryAlignment` static data member might be surprising at first. In most cases the alignment of **`Vector<T>`** will be equal to `MemoryAlignment`. However, as discussed in Section 4.1.1, implementations are free to use a SIMD register with different representation of the scalar entries than **`EntryType`**. In such a case, the alignment requirements for **`Vector<T>`** will be higher than $\mathcal{W}_T \times \mathcal{S}_T$ for an aligned load or store. Note that the load and store functions allow converting loads (Section 4.3.1). These functions need a pointer to memory of a type different than **`EntryType`**. Subsequently the alignment requirements for these pointers can be different. Starting with C++14 it may therefore be a good idea to declare `MemoryAlignment` as:

```
template <typename U>
static constexpr size_t MemoryAlignment = implementation_defined;
```

`IndexesFromZero()` The `IndexesFromZero()` function (line 14) returns a **`Vector<T>`** object where the entries are initialized to the successive values `{0, 1, 2, 3, 4, …}`. This constant is useful in many situations where the different SIMD lanes need to access different offsets in memory or to generate an arbitrary uniform offset vector with just a single multiplication.

3 It would suffice to define only the `size()` function and drop `size()`. Personally, I prefer to not use function calls in constant expressions. Additionally, a 50% difference in the number of characters makes `size()` preferable because it is such a basic part of using SIMD types.

4.1.3 NAMESPACE

Line 2 defines a namespace that is not part of the standard interface. A user should be able to access the natural SIMD registers and operations via the `Vector<T>` type and its typedefs `float_v`, `double_v`, ... However, consider the case of compiling two translation units with different target compiler flags. In that case `Vector<T>` could map to different SIMD widths even though they had the same type. Thus, the code would compile, link, and possibly even run—but not correctly—. The namespace turns the `Vector<T>` types into different symbols and thus ensures correct linkage of different translation units. An implementation may choose to document the target-dependent namespaces, as discussed in Section 4.9.2.

4.1.4 SIMD TYPE ALIASES

The code on lines 20–25 declares handy aliases for the `Vector<T>` class template. The intent of these aliases is to make SIMD vector code more concise and recognizable.

There is a design decision here: whether to use the types `char`, `short`, `int`, `long`, and `long long` or the `int8_t`, `int16_t`, `int32_t`, and `int64_t` typedefs. The problem with the latter list is that these types are optional. Thus, the definition of `int32_v` (≡ `Vector<int32_t>`) is optional, too. Since the `intN_t` typedefs must map to one of the fundamental types in the first list of types, definition of the SIMD Types with the fundamental types of the first list is more general.

It is a sensible choice to additionally declare `typedef`s for `intN_v` in the presence of `intN_t` typedefs.

4.2 SIMD VECTOR INITIALIZATION

The interface for initialization (excluding loads (covered in Section 4.3) and gathers (covered in Section 4.8)) is shown in Listing 4.2. The Vc vector types are not POD[4] types because the class interface needs full control over the implicit and explicit initialization methods (and because there is no guarantee about the POD-ness of `VectorType`). The decision for what constructors to implement follows from syntactical and semantical compatibility with the builtin arithmetic type `EntryType`. Thus, the expressions in Listing 4.3 must compile and behave analogous to the corresponding fundamental types.

4 Plain Old Data

```
// init to zero
Vector();

// broadcast with implicit conversions
Vector(EntryType);

// disambiguate broadcast of 0 and load constructor
Vector(int);  // must match exactly

// implicit conversion from compatible Vector<T>
template <typename U>
  requires ImplicitConversionAllowed<U, EntryType>()
Vector(Vector<U>);

// static_cast from vectors of possibly (depending on target)
// different size (dropping values or filling with 0 if the size is
// not equal)
template <typename U>
  requires ExplicitConversionAllowed<U, EntryType>()
explicit Vector(Vector<U>);
```

Listing 4.2: Initialization and conversion constructors for `Vector<T>`.

```
double_v a0{}, a1();  // zero-initialized

float_v  b = 0, c = 2.f;
short_v  d = -1;           // -1
ushort_v e = -1;           // numeric_limits<unsigned short>::max()
int_v    f = short(-1);   // -1
uint_v   g = short(-1);   // numeric_limits<unsigned int>::max()

ushort_v h = d;  // numeric_limits<unsigned short>::max()
int_v    i = g;  // implementation-defined value for i

float_v  j = static_cast<float_v>(a);
double_v k = static_cast<double_v>(d);
```

Listing 4.3: A few statements that are valid initialization expressions, if the builtin scalar types were used. They must work equally well with the `Vector<T>` types (as shown).

DEFAULT CONSTRUCTOR The default constructor on line 2 creates a zero-initialized object. The constructor may not keep the object uninitialized. Because, if the expression `T()` is used with a fundamental type, a "prvalue of the specified type, which is value-initialized" [48, §5.2.3] is created. The term "value-initialized" implies "zero-initialized" for fundamental types.

DESTRUCTOR OR COPY/MOVE CONSTRUCTOR There is no need for a destructor and explicitly declared copy and/or move constructors as long as the vector type does not use external storage.[5] There might be a need for these functions if the `Vector<T>` type is used as a handle to remote data, for instance on an accelerator card. Such an implementation needs to be able to add the destructor and copy/move constructors and assignment operators, though.

4.2.1 BROADCASTS

The constructor on line 5 declares an implicit conversion from any value of a type that implicitly converts to `EntryType`. This means that in places where a variable of type `Vector<T>` is expected, a variable of type `T` works as well. The constructor then broadcasts the scalar value to all entries of the SIMD vector. This kind of implicit conversion makes it very easy and natural to use numeric constants in SIMD code.

The constructor on line 8 is a special case of the preceding broadcast constructor on line 5. This constructor is required because initialization with the literal `0` is ambiguous otherwise. The load constructor (see Section 4.3 line 8 of Listing 4.7) and the `Vector(EntryType)` constructor match equally well, with just a single implicit type conversion. If `EntryType` is `int`, then this extra constructor overload on line 8 must be removed from overload resolution, because otherwise the signatures of the constructors on lines 5 and 8 are equal. For all the other `Vector<T>` types the `Vector(int)` constructor must not participate in overload resolution except when the argument to the constructor is exactly an `int`. Otherwise the expression `short_v v(1u)` would be ambiguous. This can be implemented with a template parameter which must be deduced to exactly be an `int` using an additional `enable_if` parameter:

```
template <typename U>
Vector(U a, typename enable_if<is_same<U, int>::value &&
                               !is_same<U, EntryType>::value,
                           void *>::type = nullptr);
```

5 See Section 8.2.1 for why a trivial copy constructor makes an important difference.

```cpp
template <typename A, typename B>
concept bool ImplicitConversionAllowed() {
  return is_integral<A>::value && is_integral<B>::value &&
         is_same<conditional_t<is_signed<A>::value,
                                make_unsigned_t<A>, make_signed_t<A>>,
                 B>::value;
}
```

Listing 4.4: Possible implementation of the **ImplicitConversionAllowed** concept.

4.2.2 SIMD VECTOR CONVERSIONS

The fundamental arithmetic types implicitly convert between one another. (Not every such conversion is value-preserving, which is why some compilers emit warnings for type demotions, and why brace-initialization with a narrowing conversion is ill-formed.) Conversions should work in the same manner for SIMD types. However, there is no guarantee that the number of scalar entries in a SIMD vector type is equal to the number of entries in a different type. Therefore, the conversions between **Vector**<**T**> types are split into implicit and explicit conversions. The intent is expressed with `requires` expressions (the Concepts Lite C++ extension that is on track for C++17 [77]) and can just as well be implemented with `enable_if`.

It is important that code written with the **Vector**<**T**> types is as portable as possible. Therefore, implicit casts may only work if $\mathcal{W}_{\mathtt{T}} = \mathcal{W}_{\mathtt{U}}$ holds on every possible target system. There is no real guarantee for this to work with any type combination. It is a reasonable assumption, though, that $\mathcal{W}_{\mathtt{T}} = \mathcal{W}_{\mathtt{make\_signed\_t<T>}}$ for any unsigned integral type **T** (since **make_signed_t<T>** "occupies the same amount of storage" as **T** [48, §3.9.1]). Therefore, the `ImplicitConversionAllowed` concept (line 12) must check for both types to be integral and to differ only in signedness (Listing 4.4).

If only these implicit casts were allowed, then the interface would be too restrictive. The user needs to be able to convert between SIMD vector types that possibly have a different number of entries. The constructor on line 20 therefore allows all remaining conversions not covered by the preceding constructor. Since the constructor is declared `explicit` it breaks with the behavior of the builtin arithmetic types and only allows explicit casts (such as `static_cast` or explicit constructor calls).

It would certainly be possible to define additional guaranteed $\mathcal{W}_{\mathtt{T}}$ relations by requiring implementations to implement some vector types with multiple registers. As a matter of fact, this is how I initially implemented Vc: $\mathcal{W}_{\mathtt{int}} = \mathcal{W}_{\mathtt{float}}$ was guaranteed. This was easy to support with only SSE and Larrabee (now Xeon Phi) implementations, but required two SSE **int** vectors to implement the AVX target. This is due to the initial AVX revision not including instruction support for inte-

```cpp
float_v f(float_v x) {
  float_v r;
  for (size_t i = 0; i < float_v::size(); i += double_v::size()) {
    r = r.shiftIn(
        double_v::size(),
        static_cast<float_v>(g(static_cast<double_v>(x.shifted(i)))));
  }
  return r;
}
```

Listing 4.5: Sample code for portably calling the function `g(double_v)` on a full `float_v`.

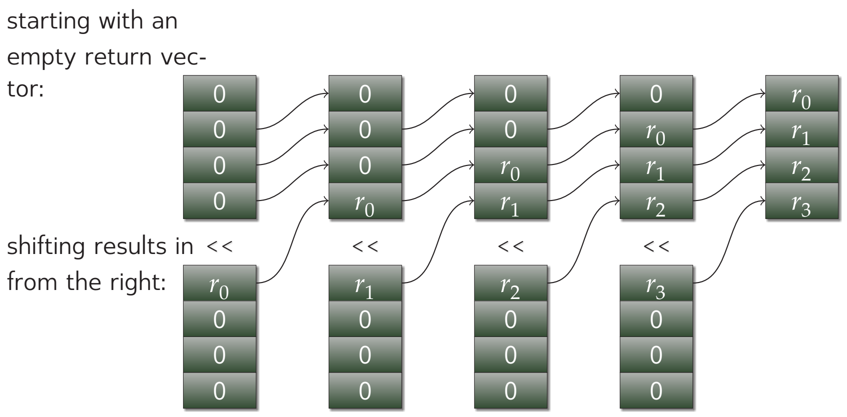

starting with an empty return vector:
shifting results in from the right:
<<
0
r_0
r_1
r_2
r_3

Figure 4.1: Vector filling algorithm used in Listing 4.5

ger vectors of 256 bits. However, conversions between different integer types and floating-point types are very important to many algorithms and therefore must be supported.

The conversion constructor on line 20 converts $n = \min(\mathcal{W}_\mathtt{T}, \mathcal{W}_\mathtt{U})$ values for a conversion from **Vector<T>** to **Vector<U>**. The remaining $n_0 = \max(0, \mathcal{W}_\mathtt{U} - n)$ entries in the target variable are set to zero. This implies that user code that wants to portably implement an algorithm, which needs to convert between vectors of possibly different widths, the use of vector types as defined here is cumbersome. One would have to employ a pattern such as shown in Listing 4.5.

The idea of the code in Listing 4.5 is visualized in Figure 4.1. However, this code does not qualify as an intuitive solution. A developer will rather think of the subscript operator for accomplishing the task. Subscripting for a target-dependent number of entries in the vector requires an even more awkward and non-obvious interface. Listing 4.6 shows a possible syntax for a portable subscripting solution. It thus becomes clear that a better solution for the cast issue needs to be provided. One possible solution uses a new cast function (`simd_cast`) that can cast from mul-

```cpp
float_v f(float_v x) {
  float_v r;
  for (size_t i = 0; i < float_v::size(); i += double_v::size()) {
    r[{i, double_v::size()}] = static_cast<float_v>(
        g(static_cast<double_v>(x[{i, double_v::size()}])));
  }
  return r;
}
```

Listing 4.6: Syntax idea for implementing Listing 4.5 via the subscript operator. The argument to the subscript operator consists of a start offset and a length, thus allowing arbitrary partial assignment/extraction. Since the subscript operator only supports a single parameter the argument must be converted from an initializer list.

tiple `Vector<T>` to one `Vector<U>` or from one `Vector<U>` to multiple `Vector<T>`.[6] An even more convenient solution builds upon these casts and the `Vector<T>` type to create a `SimdArray<T, N>` abstraction for composing multiple `Vector<T>` into one type (cf. Chapter 7).

4.3 LOADS AND STORES

The vector types need an interface for loading and storing SIMD vectors from/to memory. In contrast to the primary motivation of providing the same syntax and semantics as for fundamental types, these functions have no equivalent in their underlying fundamental types `T`. The load & store functions are required because they are a portable and efficient interface for converting between arrays of a fundamental scalar type and vector types. Without load & store functions, data could not reasonably be converted in a portable fashion: All input and output of data to a vectorized algorithm would be required to exclusively use the `Vector<T>` types. Obviously, this would be an unrealistic requirement for the majority of applications.

Nevertheless, loads and stores are an unfortunate requirement that should rather be eliminated from the set of required operations user code has to call. There are different ideas to hiding loads and stores behind abstractions on top of SIMD types and standard containers. These abstractions still build upon the load & store functions in the `Vector<T>` interface, though. Chapter 9 describes one of the ideas.

Listing 4.7 shows the interface for loads and stores for the `Vector<T>` types. These functions convert $\mathcal{W}_\mathtt{T}$ consecutively stored scalar objects of type `T` to one object of type `Vector<T>` and back. Thus, the start address (pointer to the first scalar object) and the type of the scalar objects are sufficient to fully characterize

6 Such a `simd_cast`function is implemented in Vc, supporting `Vector<T>`, `Mask<T>` (cf. Chapter 5), `SimdArray<T, N>`, and `SimdMaskArray<T, N>` (cf. Chapter 7). This shows that it is possible to define and implement this function generically.

```cpp
// load member functions
void load(const EntryType *mem);
template <typename Flags> void load(const EntryType *mem, Flags);
template <typename U, typename Flags = UnalignedT>
void load(const U *mem, Flags = Flags());

// load constructors
explicit Vector(const EntryType *mem);
template <typename Flags>
explicit Vector(const EntryType *mem, Flags flags);
template <typename U, typename Flags = UnalignedT>
explicit Vector(const U *mem, Flags flags = Flags());

// store functions
void store(EntryType *mem) const;
void store(EntryType *mem, MaskType mask) const;
template <typename Flags>
void store(EntryType *mem, Flags flags) const;
template <typename Flags>
void store(EntryType *mem, MaskType mask, Flags flags) const;
template <typename U, typename Flags = UnalignedT>
void store(U *mem, Flags = Flags()) const;
template <typename U, typename Flags = UnalignedT>
void store(U *mem, MaskType mask, Flags = Flags()) const;
```

Listing 4.7: Declaration of the load and store functions.

```cpp
void Vector<T>::load(const U *mem) {
  for (size_t i = 0; i < size(); ++i) {
    (*this)[i] = static_cast<T>(mem[i]);
  }
}
void Vector<T>::store(U *mem) const {
  for (size_t i = 0; i < size(); ++i) {
    mem[i] = static_cast<U>((*this)[i]);
  }
}
```

Listing 4.8: The semantics of a converting load. A concrete implementation will use vector loads and conversions, of course.

the required load or store operation. (The case of distributed scalars is handled by gather and scatter functions, which are described in Section 4.8.)

4.3.1 CONVERTING LOADS AND STORES

Some SIMD hardware can convert between different data types without extra runtime overhead when executing a load or store instruction [46]. Therefore, and because it is very convenient for writing portable conversion code, the load & store functions provide a generic variant that can access arrays of different scalar types. Semantically, these functions behave as described in Listing 4.8. Thus, $\mathcal{W}_{\mathtt{T}}$ values of type `U` are converted with load/store functions in `Vector<T>`, independent of $\mathcal{W}_{\mathtt{U}}$, in contrast to the converting constructor in Section 4.2.2.

Not all conversions are equally efficient in terms of hardware support. However, for reasons of portability, the full set of conversions between fundamental arithmetic types is made available through these functions.

4.3.2 LOAD/STORE FLAGS

SIMD hardware makes a difference between aligned and unaligned vector loads and stores (cf. Section 1.3.1). Additionally, most algorithms can be optimized if the developer can hint at the temporal usage of the data.[7] The alignment can, in theory, be determined from the start address, and thus would not require additional specification in the function call. However, since the alignment can only be determined from the pointer value at runtime, such a check would incur a penalty. Using unaligned load/store instructions unconditionally would be more efficient than checking the alignment of the pointer. An unaligned load/store instruction in hardware can do the alignment test much more efficiently. Therefore, per default, the load/store functions translate to unaligned load/store instructions.

4.3.2.1 ALIGNMENT

If the user can guarantee alignment, a tag type can be used as last argument to select the optimized load/store instructions at compile time, without any runtime overhead. It is important that the API is built via a `template` and tag type, rather than a boolean (or enum) function argument. A boolean function argument cannot guarantee compile-time optimization. Especially, such an API would allow passing a non-constant expression as flag variable which cannot be optimized at all. Via the tag type the user of the API is required to provide a constant expression and thus decide between aligned or unaligned memory access when (s)he writes the code.

4.3.2.2 NON-TEMPORAL ACCESS

Loads and stores can be further optimized for non-temporal accesses. Many data-parallel algorithms use a streaming pattern, where the input and/or output memory locations are used only once. Therefore, this data should not evict other data, which might be used repeatedly in the algorithm, from the CPU caches. The load-/store functions in Vc can therefore be called with the `Vc::Streaming` tag type. This tag hints to the `Vector<T>` implementation that the data can be moved past the caches. (Most CPUs can use specialized load and store buffers for streaming loads and stores.) If the requested load/store operation cannot be executed as a streaming variant, the implementation will silently fall back to a non-streaming variant.

7 Non-temporal load/store hints tell the CPU that the given memory addresses are referenced only once and memory transfers may bypass the cache hierarchy.

Streaming stores executed with the `Vc::Streaming` tag may use non-globally ordered stores if the target CPU supports this. Thus, two stores to the same memory location, where at least one is a streaming store, have undefined behavior unless a memory fencing operation is called between the stores. This allows to reach the highest store throughput, but requires a good understanding of the implications when used by a developer.

4.3.2.3 PREFETCHING

A last flag that I implemented for the load/store functions makes prefetching in loops significantly simpler. By adding the `Vc::PrefetchDefault` tag type, the `Vector` implementation is asked to emit software prefetch instructions for a target-dependent predefined stride. Thus, a call to

```
float_v(memory, Vc::Aligned | Vc::PrefetchDefault)
```

may result in up to three instructions being called, one of which is the load instruction. In addition, prefetch instructions for the lowest level cache and second lowest level cache may be called. These prefetches are called with a predefined offset to the `memory` address that is passed to the load function.

The prefetch flag is therefore a shorthand to prefetching explicitly in many loops. However, not all loops require the same prefetch stride lengths, which is why instead of the predefined strides the user may also set the strides explicitly. In almost all cases, a developer adds prefetches after the program or component is already working and is only modified for speed optimizations. The developer then determines the prefetch strides through intuition and/or trial and error.

Note that prefetches only need to be called once for any address inside one cache line. Thus, two subsequent loads/stores to neighboring SIMD vectors may result in more software prefetch instructions than necessary. This depends on the ratio of the cache line size to the vector register size. As this ratio is target dependent, the API appears to introduce a portability issue in this case. There is no easy solution from the load/store interface side. However, the compiler is, in theory, able to drop the superfluous prefetch instructions.[8]

8 This is possible if the relative difference between prefetch instructions is considered by the compiler. It could apply an algorithm that keeps the first prefetch call and drops every subsequent prefetch call that would reference the same cache line as a previous call.

```
1   Vector &operator++();
2   Vector  operator++(int);
3   Vector &operator--();
4   Vector  operator--(int);
5
6   MaskType operator!() const;
7   Vector operator~() const;
8   Vector operator+() const;
9   Vector operator-() const;
```

Listing 4.9: Declaration of unary operators.

4.4 UNARY OPERATORS

The unary operators (increment, decrement, logical negation, one's complement, unary +, and unary -) behave as $\mathcal{W}_{\mathtt{T}}$ applications of the operator to the scalar values in the vector. However, there is an API issue that results from *integral promotion*, which is applied to the operand of unary +, unary -, and one's complement. Integral promotion leads to operands of builtin integral types smaller than `int`/`unsigned int` getting promoted to `int` or `unsigned int` before the operator is applied. This implies that `ushort_v` and `short_v` would have to return `int_v` from unary +, unary -, and one's complement. However, since $\mathcal{W}_{\mathtt{short}} = \mathcal{W}_{\mathtt{int}}$ does not hold for most targets, this is not possible (except if the return type were `SimdArray<int, short_v::size()>`[9]). For the fundamental integral types, integral promotion to `int` normally does not cause a performance hit because `int` is defined to be the natural integer type of the target CPU. Though, for vector types, integral promotion may require more overhead than is acceptable. Therefore, the SIMD vector types do not perform integral promotion. This is also the case for binary operators (cf. Section 4.5). On the other hand, integral promotion may make much more sense for the `SimdArray<T, N>` types.

The declaration of the interface for the unary operators is shown in Listing 4.9. As discussed above, the return types of the functions on lines 7–9 do not follow the integral promotion rules. These operators can therefore lead to subtle differences compared to scalar code, such as the example in Listing 4.10 demonstrates. The assertion on line 7 fails because the builtin compare operator performs integral promotion, promoting the right hand side of the comparison to `int`. However, while `b` holds the value −40000, `w` holds $\mathcal{W}_{\mathtt{short}}$ values of $2^{16} - 40000 = 25536$. Conversion of one value in `w` to `int` does not change that value. Thus, line 7 compares `-40000 == 25536`.

9 See Chapter 7.

```
unsigned short a = 40000;
auto b = -a; // decltype(b) == int

ushort_v v = 40000;
auto w = -v; // decltype(w) == ushort_v

assert(b == w[0]); // this fails
```

Listing 4.10: Subtle differences between scalar code and vector code, because of differences in integer promotion.

4.5 BINARY OPERATORS

Binary operators express arithmetic, comparison, bitwise, and shift operations. They implement the central part of the SIMD functionality by executing $\mathcal{W}_T$ operations in parallel on the SIMD vector entries. If the `EntryType` is not of integral type, the bitwise, shift, and modulo operators need to be ill-formed. They would certainly be implementable, but since the builtin non-integral types do not implement these operators, the SIMD types follow the same semantics.

The interface for these operators is shown in Listing 4.11. In this form of declaration, the compiler will allow the right-hand operand to be implicitly converted via a non-explicit conversion constructor. Thus, conversions from integer-vectors of differing signedness and broadcasts from scalars would be possible. However, the resulting type would solely be determined by the type of the left-hand operand. Consequently, `int_v() + uint_v()` would result in an `int_v`, whereas `uint_v() + int_v()` would result in a `uint_v`. Also, `int_v() + 1.f` would compile and result in an `int_v`, whereas any operation with a scalar value on the left-hand side (such as `1.f + float_v()`) would not compile at all. Thus, there is a need for further refinement of the binary operators, which can be done via non-member operators.

4.5.1 GENERIC NON-MEMBER BINARY OPERATORS

The definition of two of the non-member binary operators (one arithmetic and one comparison operator) is shown in Listing 4.12. There is only a slight difference in the return type between comparison operators and the remaining binary operators. Compares obviously must return a mask type and therefore require the `Vector<T>::MaskType` return type. The operator's implementation simply forwards to the member operators using the same type for the left and right operands. The evaluation of this type does all the magic. Especially important is the fact that if `TypesForOperator<L, R>` leads to a substitution failure no error is emitted, but the operator is silently removed from the candidate list (SFINAE[10] [81]).

10 Substitution Failure Is Not An Error

```cpp
  Vector operator* (Vector x) const;
  Vector operator/ (Vector x) const;
  Vector operator+ (Vector x) const;
  Vector operator- (Vector x) const;

  MaskType operator==(Vector x) const;
  MaskType operator!=(Vector x) const;
  MaskType operator>=(Vector x) const;
  MaskType operator<=(Vector x) const;
  MaskType operator> (Vector x) const;
  MaskType operator< (Vector x) const;

private:
  // Use SFINAE to disable the operator if EntryType is not integral:
  // 1. Variant, for Vector arguments
  template <typename U>
  using ReturnVectorIfIntegral =
      typename std::enable_if<std::is_convertible<U, Vector>::value &&
                                  std::is_integral<EntryType>::value,
                              Vector>::type;
  // 2. Variant, for int arguments
  template <typename U>
  using ReturnVectorIfIntInt =
      typename std::enable_if<std::is_convertible<U, int>::value &&
                                  std::is_integral<EntryType>::value,
                              Vector>::type;

public:
  template <typename U> ReturnVectorIfIntegral<U> operator%(U x) const;

  template <typename U> ReturnVectorIfIntegral<U> operator&(U x) const;
  template <typename U> ReturnVectorIfIntegral<U> operator|(U x) const;
  template <typename U> ReturnVectorIfIntegral<U> operator^(U x) const;

  template <typename U> ReturnVectorIfIntegral<U> operator>>(U x) const;
  template <typename U> ReturnVectorIfIntegral<U> operator<<(U x) const;
  template <typename U> ReturnVectorIfIntInt  <U> operator>>(U x) const;
  template <typename U> ReturnVectorIfIntInt  <U> operator<<(U x) const;
```

Listing 4.11: Declaration of binary operators.

```cpp
template <typename L, typename R>
inline TypesForOperator<L, R> operator+(L &&x, R &&y) {
  using V = TypesForOperator<L, R>;
  return V(std::forward<L>(x)) + V(std::forward<R>(y));
}
// ...

template <typename L, typename R>
inline typename TypesForOperator<L, R>::MaskType operator==(L &&x,
                                                            R &&y) {
  using V = TypesForOperator<L, R>;
  return V(std::forward<L>(x)) == V(std::forward<R>(y));
}
// ...
```

Listing 4.12: Non-member operators for the Vc SIMD vector types.

Thus, **`TypesForOperator<L, R>`** determines every aspect of which binary operators are supposed to compile or throw an error and which implicit conversions are involved.

The **`TypesForOperator<L, R>`** type (see Listing 4.13, line 56) in Vc is defined as an alias template for the **`TypesForOperatorInternal`** `struct`. The alias template simplifies **`TypesForOperatorInternal`** by swapping the types **`L`** and **`R`** if **`L`** is a non-vector type. Therefore, the first template argument to **`TypesForOperator-Internal`** is guaranteed to be a SIMD vector type, unless neither **`L`** nor **`R`** is a SIMD vector type. The third template argument (a boolean) provides a simple mechanism to specialize the `struct` for type combinations where a substitution failure should result (and thus inhibit the binary operator to participate in overload resolution). The conditions for this are simply that the non-member operators may only participate in overload resolution for type combinations that involve at least one vector type and where **`L`** and **`R`** are not equal, in which case the operator would lead to an ambiguity with the member operator in the vector class.

The `struct` on line 48 defines an empty type for operands where the operators are not supposed to match. Note that the `struct` does not contain the member type **`type`**, thus leading to the intended substitution failure. The `struct` on line 51 specializes for the case where **`V`** is a SIMD vector type, the operand types are different, **`W`** is convertible to a SIMD vector type, and the combination of types yields valid implicit conversions. Since not all conversions to SIMD vector types or between vector types are implicit, the binary operator may not invoke such a conversion and turn an explicit conversion to an implicit one. This is determined via the `isValidOperandTypes` function defined on line 35. For all allowed type combinations **`V`** and **`W`** the member type **`type`** on line 52 determines the SIMD vector type to use as return type and for conversion of the operands before calling the member operator.

By additionally declaring operator overloads that are `!isValidOperandTypes` as deleted, the interface catches incorrect use and gives some hint to the user why the code does not compile (Listing 4.14). Currently C++ only allows to encode an explanation in the type name. Appendix I describes the issue of custom diagnostics for ill-formed function overloads in more detail and suggests a simple extension to the standard to improve the diagnostic output.[11]

The `isValidOperandTypes` function ensures that the following type combinations for the operands are invalid:

11 Using a `static_assert` for improved error messages is also possible here and it can be used to explain the error directly, thus making correcting errors in the usage of the interface easier. On the other hand, with a `staticāssert`, a trait that checks whether a binary operator for two given operands is defined will return a positive answer even though an actual call would fail to compile because of the static assertion. (see Appendix I for the details)

```cpp
template <typename T, bool> struct MakeUnsignedInternal;
template <typename T> struct MakeUnsignedInternal<Vector<T>, true> {
  using type = Vector<typename std::make_unsigned<T>::type>;
};
template <typename T> struct MakeUnsignedInternal<Vector<T>, false> {
  using type = Vector<T>;
};
template <typename Test, typename T>
using CopyUnsigned = typename MakeUnsignedInternal<
    T, isIntegral<T>() && isUnsigned<Test>()>::type;

template <typename From, typename To>
constexpr bool isNarrowingFloatConversion() {
  return is_floating_point<From>::value &&
         (is_integral<To>::value || (is_floating_point<To>::value &&
                                     sizeof(From) > sizeof(To)));
}

template <typename T> static constexpr bool convertsToSomeVector() {
  return is_convertible<T, double_v>::value ||
         is_convertible<T, float_v>::value ||
         is_convertible<T, int_v>::value ||
         is_convertible<T, uint_v>::value ||
         is_convertible<T, short_v>::value ||
         is_convertible<T, ushort_v>::value;
}

template <typename V, typename W>
constexpr bool participateInOverloadResolution() {
  return isVector<V>() && !is_same<V, W>::value &&
         convertsToSomeVector<W>();
}

template <typename V, typename W>
constexpr enable_if<isVector<V>(), bool> isValidOperandTypes() {
  using type = CopyUnsigned<W, V>;
  return isVector<W>() ? (is_convertible<V, W>::value ||
                          is_convertible<W, V>::value)
                       : (is_convertible<W, type>::value &&
                          !isNarrowingFloatConversion<
                              W, typename type::EntryType>());
}

template <
    typename V, typename W,
    bool VectorOperation = participateInOverloadResolution<V, W>() &&
                           isValidOperandTypes<V, W>()>
struct TypesForOperatorInternal {};

template <typename V, typename W>
struct TypesForOperatorInternal<V, W, true> {
  using type = CopyUnsigned<W, V>;
};

template <typename L, typename R>
using TypesForOperator = typename TypesForOperatorInternal<
    decay_t<conditional_t< isVector<L>(), L, R>>,
    decay_t<conditional_t<!isVector<L>(), L, R>>>::type;
```

Listing 4.13: The traits the non-member binary operators need for SFINAE and return type evaluation.

```cpp
template <
    typename V, typename W,
    bool IsIncorrect = participateInOverloadResolution<V, W>() &&
                       !isValidOperandTypes<V, W>()>
struct IsIncorrectVectorOperands {};
template <typename V, typename W>
struct IsIncorrectVectorOperands<V, W, true> {
  using type = void;
};

template <typename L, typename R>
using Vc_does_not_allow_operands_to_a_binary_operator_which_can_have_\
different_SIMD_register_sizes_on_some_targets_and_thus_enforces_\
portability =
    typename IsIncorrectVectorOperands<
        Traits::decay<Conditional<isVector<L>(), L, R>>,
        Traits::decay<Conditional<!isVector<L>(), L, R>>>::type;

template <typename L, typename R>
Vc_does_not_allow_operands_to_a_binary_operator_which_can_have_\
different_SIMD_register_sizes_on_some_targets_and_thus_enforces_\
portability<L, R> operator+(L &&, R &&) = delete;
```

Listing 4.14: Declaration of explicitly deleted operators for improved diagnostics on incorrect usage.

- If both operands are SIMD vectors, at least one of them must be implicitly convertible to the other type.
- If one operand is a scalar type, then an implicit conversion from the scalar type to the return type must be possible.
- Furthermore, a conversion from scalar type to vector type may not lead to a narrowing conversion from a floating point type. This essentially forbids `float_v` $\times$ `double` because the `double` operand would have to be converted to the narrower single-precision `float` type. On the other hand `double_v` $\times$ `float` does not require a narrowing conversion and therefore works.

The return type is determined via the `CopyUnsigned<Test, T>` alias template on line 9. The rules are as follows:

- The return type is the unsigned variant of `T` if `T` is integral and `Test` is an unsigned type.
- Otherwise the return type is `T`.

Thus, if one operand is an unsigned integer vector or scalar and the other operand is a signed integer vector or scalar, then the operands are converted to the corresponding unsigned integer vector. However, in contrast to the semantics of builtin integer types, no full integer promotion is applied, leaving $\mathcal{S}_T$ of the vector entries,

```
Vector &operator*= (Vector<T> x);
Vector &operator/= (Vector<T> x);
Vector &operator%= (Vector<T> x);
Vector &operator+= (Vector<T> x);
Vector &operator-= (Vector<T> x);

Vector &operator&= (Vector<T> x);
Vector &operator|= (Vector<T> x);
Vector &operator^= (Vector<T> x);
Vector &operator<<=(Vector<T> x);
Vector &operator>>=(Vector<T> x);
Vector &operator<<=(int x);
Vector &operator>>=(int x);
```

Listing 4.15: Declaration of assignment operators.

and thus $\mathcal{W}_T$ unchanged. It follows that **`int_v`** × **`unsigned int`** yields **`uint_v`** and **`short_v`** × **`unsigned int`** yields **`ushort_v`**. The latter implicit conversion from **`unsigned int`** to **`ushort_v`** is unfortunate, but since **`short_v`** `+ 1` should be valid code and return a **`short_v`**, it is more consistent to also convert **`unsigned int`** implicitly to **`short_v`** or **`ushort_v`**.

The non-member operators were explicitly designed to support operator calls with objects that have an implicit conversion operator to a SIMD vector type. This is possible by leaving **`W`** less constrained than **`V`** (in Listing 4.13).

4.5.2 OPTIMIZING VECTOR × SCALAR OPERATIONS

Note the shift operator overloads for an argument of type **`int`** on lines 38–37 in Listing 4.11. This touches a general issue that is not fully solved with the binary operators interface as declared above: Some operations can be implemented more efficiently if the operator implementation knows that one operand is a scalar or even a literal. A scalar operand would be converted to a SIMD vector with equal values in all entries via the non-member binary operators (which Vc therefore does not define for the shift operators).

The issue is certainly solvable, and a topic for future research. A possible solution could be not to call `V::operator···(V)` from the non-member operators and instead call a template function such as

```
template <typename V, typename L, typename R>
V execute_operator_add(L &&, R &&)
```

. This function could then be overloaded such that one overload implements Vector + Vector and the other overload implements Vector + Scalar.

4.6 COMPOUND ASSIGNMENT OPERATORS

Apart from simple assignment, C++ supports compound assignment operators that combine a binary operator and assignment to the variable of the left operand. The

```
EntryReference operator[](size_t index);
EntryType operator[](size_t index) const;
```

Listing 4.16: Declaration of the subscript operators for scalar access.

standard defines the behavior "of an expression of the form E1 op = E2 [...] equivalent to E1 = E1 op E2 except that E1 is evaluated only once." [48, §5.17] Thus, the simplest implementation calls the binary operator and the assignment operator.

However, compound assignment operators could do more than the combination of binary operator and assignment operator. The additional constraint of compound assignment operators, that the result of the binary operator needs to be converted back to the type of the left operand, allows any scalar arithmetic type as operand. This may be best explained with an example. Consider

```
short_v f(short_v x) {
  return x *= 0.3f;
}
```

with SSE where $\mathcal{W}_{\texttt{short}} = 2 \cdot \mathcal{W}_{\texttt{float}}$. In this case `Vector<short>::operator*=(float)` can be implemented as two `float_v` multiplications with the low and high parts of `x`. For binary operators this is not implemented because the return type would have to be something like `std::array<float_v, 2>` or `std::tuple<float_v, float_v>`. However, for compound assignment operators this return type problem does not exist because the two `float_v` objects will be converted back to a single `short_v`.

At this point only the more restrictive compound assignment operators are implemented in Vc and shown in Listing 4.15. The `SimdArray<T, N>` type (cf. Chapter 7) could make this easier to implement. The exact details of the necessary interface are a topic for future research.

4.7 SUBSCRIPT OPERATORS

Subscripting SIMD vector objects is a very important feature for adapting scalar code in vectorized codes. Subscripting makes mixing of scalar and vector code intuitively easy (though sometimes at a higher cost than intended).

However, while subscripting is desirable from a users point of view, the C++ language standard makes it close to impossible. The issue is that the non-const subscript operator needs to return an lvalue reference to the scalar type (`EntryReference`) and assignment to this lvalue reference needs to modify the SIMD vector, which is of type `VectorType`. This requires aliasing two different types onto the same memory location, which is not possible with standard C++. Even a `union` does not solve the issue because aliasing via `union`s is only well-defined for layout-compatible types.

```
EntryType operator[](size_t index) const {
  EntryType mem[size()];
  store(mem);
  return mem[index];
}
```

Listing 4.17: The offset in a SIMD register via `operator[]` is defined by the memory order.

Therefore, the return type of the non-const subscript operator is *implementation-defined* (see Listing 4.16). Most compilers provide extensions to standard C++. One popular extension is explicit aliasing via unions. However, there are also other ways of allowing explicit aliasing, such as GCC's `may_alias` attribute. If Vc is compiled with GCC then the return type of the non-const subscript operator will be **`EntryType`** with the `may_alias` attribute.

The const subscript operator intentionally returns a prvalue and not a const lvalue reference. With a reference, read-only access would require the compiler to also violate the strict aliasing rules. And it is a very rare use-case to store a reference to the return value from the const subscript operator and then modify the object through a non-const reference. If the user wants to have a (mutable or immutable) reference to an entry in the SIMD vector (s)he can (and must) use the non-const subscript operator. Finally, as noted in Section 4.1.1, the **`EntryReference`** type is not necessarily the same as `EntryType&`. Therefore, in order to actually return **`EntryType`**, the generic definition of the const operator needs to return a prvalue.

The effect of the **`const`** subscript operator is shown in Listing 4.17. Most importantly, this defines the indexing order of the values in the SIMD vector: Access to the value at offset `i` via the subscript operator yields the same value a store and subsequent array access at offset `i` produces. Of course, the same indexing order must be used for the non-const subscript operator.

4.8 GATHER & SCATTER

A gather operation in SIMD programming describes a vector load from a given base address and arbitrary (positive) offsets to this address. The scatter operation is the reverse as a vector store. (see Figure 4.2)

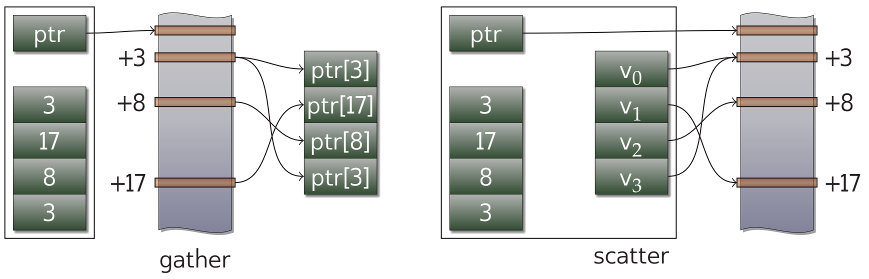


Figure 4.2: Graphical representation for gather and scatter operations. Both operations needs a base pointer (`ptr`) and a vector of offsets to this pointer (`{3, 17, 8, 3}`). This is used to calculate the addresses in memory: For the gather operation, these values from memory are read into the corresponding lanes of the return vector. For the scatter operation a third input—a vector of values—is written into the memory locations.

```
struct iovec {
  void *iov_base; // Pointer to data
  size_t iov_len; // Length of data
};
ssize_t readv (int fildes, const struct iovec *iov, int iovcnt);
ssize_t writev(int fildes, const struct iovec *iov, int iovcnt);
```

Listing 4.18: Scatter/gather functions in POSIX [79].

4.8.1 PRIOR ART

Portable and intuitive gather & scatter APIs—in terms of function calls—do not have an obvious solution. The most interesting prior art is the syntax of **`valarray`** gather/scatter (Section 4.8.1.4).

4.8.1.1 POSIX

The Open Group Base Specifications, Issue 6, IEEE Std 1003.1 [79] specify `readv` and `writev` in POSIX 1003.1 - 2004 (Listing 4.18). `readv` can be used to scatter from a file descriptor into a given number of arbitrarily sized buffers. `writev` does the reverse of `readv`. Note that the functions are not type-safe (the **`void*`** erases the type information), which is fine for file descriptor I/O but not for a C++ API for SIMD vector types. Furthermore, the interface requires the user to create an array of pointers to the locations in memory instead of a single pointer and a vector of offsets. The POSIX gather/scatter interface thus is not a good starting point for gather/scatter for **`Vector`**`<`**`T`**`>`.

```
__m512 _mm512_i32gather_ps(__m512i index, void const *addr,
                           int scale);
__m512 _mm512_mask_i32gather_ps(__m512 v1_old, __mmask16 k1,
                                __m512i index, void const *addr,
                                int scale);
__m512 _mm512_i32extgather_ps(__m512i index, void const *mv,
                              _MM_UPCONV_PS_ENUM conv, int scale,
                              int hint);
__m512 _mm512_mask_i32extgather_ps(_m512 v1_old, __mmask16 k1,
                                   __m512i index, void const *mv,
                                   _MM_UPCONV_PS_ENUM conv, int scale,
                                   int hint);

void _mm512_i32scatter_ps(void *mv, __m512i index, __m512 v1,
                          int scale);
void _mm512_mask_i32scatter_ps(void *mv, __mmask16 k1, __m512i index,
                               __m512 v1, int scale);
void _mm512_i32extscatter_ps(void *mv, __m512i index, __m512 v1,
                             _MM_DOWNCONV_PS_ENUM conv, int scale,
                             int hint);
void _mm512_mask_i32extscatter_ps(void *mv, __mmask16 k1,
                                  __m512i index, __m512 v1,
                                  _MM_DOWNCONV_PS_ENUM conv,
                                  int scale, int hint);
```

Listing 4.19: The gather/scatter intrinsics for the Intel MIC architecture [46].

4.8.1.2 MIC INTRINSICS

Much closer in functionality to the requirements of the vector types API are the SIMD gather/scatter intrinsic functions that Intel introduced with their compiler for the MIC[12] architecture (Listing 4.19). In its simplest form the gather takes an `int` index vector, multiplies the values with the `scale` parameter (which may be 1, 2, 4, or 8) and uses these as Byte-offsets in the memory pointed to by `addr` to load 16 values. Thus `v = _mm512_i32gather_ps(index, addr, scale)` is equivalent to:

```
for (int i = 0; i < 16; ++i) {
  v[i] = *reinterpret_cast<const float *>(
             reinterpret_cast<const char *>(addr) + index[i] * scale);
}
```

In contrast to the POSIX functions, the memory regions that are read (buffers) are of fixed size ($\mathcal{S}_{\texttt{float}}$). Instead of one pointer per memory location, here a single pointer with a fixed number of offsets is used. Thus, the `_mm512_i32gather_-ps` intrinsic resembles a subscript operator applied to an array of `floats` (`v = addr[index]`). However, note that the MIC intrinsics are not type-safe: they pass the pointer as `void *` and require the caller to determine the scaling factor.

The remaining gather functions in Listing 4.19 provide additional features:

12 Many Integrated Core

```
std::valarray<std::size_t> idx = {0, 1, 2, 4};  // index array
v2.resize(4);  // sizes must match when assigning from gen subscript
v2 = v1[idx];  // from indirect array
```

Listing 4.20: Example code for a gather from **valarray** v1 to **valarray** v2 [85].

- Masked gathers allow to load fewer values from memory as determined by the corresponding bits in the mask. This allows `addr + index[i] * scale` to point to an invalid address for all `i` where `mask[i]` is `false`.
- The extended variants accept a parameter to do type conversions in addition to the load. For example, a **float** vector can be gathered from random **short**s in an array.
- The hint parameter is used to do cache optimizations, possibly marking the affected cache lines as LRU[13].

Scatter functions do the reverse of gather functions. They store a (masked) set of scalar values from a vector register to memory locations determined by the `index` and `scale` parameters.

4.8.1.3 ARRAY NOTATION

Array notation is another important interface. It is available as an extension to C/C++ with Cilk Plus [80]. With this extension the user can express operations on whole arrays with very little code. For instance `A[:] = B[:] + 5` is equivalent to `for (i = 0; i < size(A); i++) A[i] = B[i] + 5`. With this extension a gather is written as `C[:] = A[B[:]]` and a scatter accordingly as `A[B[:]] = C[:]`. The interface requires a single new concept (which is the central concept of array notation) to be type-safe, concise, and intuitively clear. This syntax also naturally expresses converting gather/scatters. Masking is possible as well by the generic masking support in the array notation syntax [43, §5.3.6], which extends the semantics of `if`/`else` statements.[14]

4.8.1.4 STL

Finally, the standard library provides related functionality in the `std::`**valarray** and `std::`**indirect_array** classes. An **indirect_array** is created if a **valarray** object is subscripted with a **valarray** <**size_t**> object. The object can then be assigned to a new or existing **valarray** object (with the right size). Listing 4.20 shows an example. The expression `v2 = v1[idx]` executes the gather operation in the assignment operator of `v2`. The reverse assignment is also possible and implements

13 Least Recently Used

14 In Cilk Plus `if`/`else` statements are extended to accept arrays of booleans. Thus, both the `if` and `else` branches can be executed. Section 5.1.1 covers this topic in depth.

```cpp
void maskedArrayGatherScatter(float *data, int_v indexes) {
  const auto mask = indexes > 0;
  float_v v(data, indexes, mask);
  v.scatter(data, indexes, mask);
}

struct S { float x, y, z; };

void structGatherScatter(S *data, int_v indexes) {
  float_v v(data, &S::x, indexes, indexes > 0);
  v.scatter(data, &S::y, indexes, indexes > 0);
  ...
}
```

Listing 4.21: Example usage of the first generation gather/scatter interface in Vc.

a scatter operation. However, these gather and scatter operations are limited to **`valarray`** objects.

4.8.2 INITIAL VC INTERFACE

The initial approach to gather and scatter interfaces in the Vc SIMD vector classes was done via member functions, and for gathers additionally via constructors. A simple gather and scatter example, using this interface, is shown in Listing 4.21. A tricky issue were gather and scatter operations on an array of a non-fundamental type, such as a struct, union, or array. In the case of an array of struct, the user needs to specify the member variable of the struct that (s)he wants to access. A possible interface for this is shown at the bottom of the example in Listing 4.21.

This interface can certainly support all features of the underlying hardware, or emulate such features on SIMD hardware that does not have the respective instruction support. However, the interface is hardly intuitive. The order of parameters does follow a logical pattern (outer array, possibly struct members, indexes, and optionally a mask as the last parameter), but even then I often had to look up the interface in the API documentation. A more intuitive interface needs to relate closer to known C++ syntax, which is something the array notation in Cilk Plus and **`valarray`** nicely demonstrate.

Furthermore, the necessary overloads to support arbitrary nesting of structs and arrays quickly get out of hand. For every possible composition of structs, unions, and arrays two (unmasked and masked) gather and scatter function overloads are required. In some situations it may be necessary to supply more than one index vector: The first indexes subscript an outer array and the offsets in the inner array are not equal for all entries but require another index vector. The gather/scatter function approach simply does not scale in that respect. The following sections will therefore present a scalable and more intuitively usable solution.

```
template <typename I>
inline auto operator[](I &&i)
    -> decltype(subscript_operator(*this, std::forward<I>(i))) {
  return subscript_operator(*this, std::forward<I>(i));
}

template <typename I>
inline auto operator[](I &&i)
    const -> decltype(subscript_operator(*this, std::forward<I>(i))) {
  return subscript_operator(*this, std::forward<I>(i));
}
```

Listing 4.22: Generic subscript member operator that forwards to a non-member function.

4.8.3 OVERLOADING THE SUBSCRIPT OPERATOR

The problems discussed above do not exist for scalar types because the subscript operator and the member access operations support arbitrary access of members in such nested data structures. Thus, the question is whether the syntax that works for scalars can be made to work for SIMD vector types as well. It is possible to overload the subscript operator with one parameter of arbitrary type, and thus also with a parameter of SIMD vector type. However, it is not possible to overload the subscript operator of existing types because C++ does not support non-member subscript operators. Thus, in order to implement gathers and scatters via the subscript operator, the array/container class needs to specifically have support for the SIMD types. On the other hand, adding a gather subscript operator directly to all container classes would make all of them depend on the declarations of the SIMD types. Luckily, there is a clean way around it that effectively creates opt-in non-member subscript operators.[15] A class simply needs the two subscript operators defined in Listing 4.22. Then, if the `subscript_operator` function is declared (with the right signature), the subscript operator can be used with the types the `subscript_operator` functions implement.

As long as the C++ standard library does not implement such a subscript operator, the container classes in the `std` namespace cannot support SIMD vector subscripting. Therefore, only a new type can implement these subscript operators. It is possible to adapt existing container classes with the **`AdaptSubscriptOperator`** class shown in Appendix H (Listing H.1) and thus create a **`Vc::vector`** type that implements `std::`**`vector`** with the additional subscript operator.[16]

15 For implementing the gather and scatter subscript operator overloads it would, of course, be better if non-member subscript operators were possible. Informal feedback from the C++ committee has been that there is interest for a concrete proposal on the issue.

16 The exception is `std::array` and other container classes that need to be POD or aggregates. `Vc::array` therefore needs to be a verbatim copy of `std::array` plus the new subscript operator.

4.8.4 A SUBSCRIPT OPERATOR FOR SIMD GATHER AND SCATTER

Once there are container classes that support subscripting with arbitrary, user-defined types, subscripting with Vc's vector types can be implemented (Listing 4.23). The requirements for the `subscript_operator` function are the following:

- It must accept only a specific subset of container classes, specifically those that use contiguous storage for its entries.
- The container may use storage on the free store. However, nested containers are constrained: Only the outermost container may use the free store.
- The container and/or its entries may be arbitrarily cv-qualified.
- It must accept a specific set of types for the index parameter.
  - Any type that implements the subscript operator and contains as many entries as the gathered vector will contain / the vector to scatter contains.
  - SIMD vectors of integral type and with `Vector<T>::size()` equal for the value vector and index vector.

The `enable_if` statement allows to implement the function such that it only participates in overload resolution if and only if …

… the `has_subscript_operator` type trait finds a usable subscript operator in the `IndexVector` type (cf. Appendix G).

… the `has_contiguous_storage` type trait determines that the `Container` type is implemented using contiguous storage for the entries in the container.[17]

… the `is_lvalue_reference` type trait determines that dereferencing the first iterator of the `Container` type returns a type that can be used to determine a pointer to the first element of the contiguous storage of the container.

Whether the `subscript_operator` function participates in overload resolution directly determines whether the generic forwarding member subscript operators in `Vc::array` and `Vc::vector` participate in overload resolution. This is due to the return type of the subscript operators, which lead to substitution failure (which is not an error) if and only if `subscript_operator` is not usable.

17 Such a trait cannot really be implemented for all I know. However, it is possible to define a list of classes and class templates that will work as expected.

```cpp
template <typename Container,
          typename IndexVector,
          typename = enable_if<
              Traits::has_subscript_operator<IndexVector>::value &&
                  Traits::has_contiguous_storage<Container>::value &&
                      std::is_lvalue_reference<decltype(
                          *begin(std::declval<Container>()))>::value>>
inline SubscriptOperation<
    typename std::remove_reference<
        decltype(*begin(std::declval<Container>()))>::type,
    typename std::remove_const<
        typename std::remove_reference<IndexVector>::type>::
        type> subscript_operator(Container &&c,
                                 IndexVector &&indexes) {
  return {std::addressof(*begin(c)),
          std::forward<IndexVector>(indexes)};
}
```

Listing 4.23: Generic `subscriptōperator` function that passes the context for a gather/scatter operation to the SubscriptOperation class.

4.8.5 a proxy type for gather/scatter

The `subscript_operator` function then returns an object of type **`SubscriptOperation`** that contains a pointer to the beginning of the container storage and a **`const`** reference to the index vector. A naïve approach would return **`Vector<T>`** directly, where **`T`** is determined by the type of the entries in the container. However, in this case converting gathers, nested subscripting, as well as any scatter operation would not be possible. Returning a proxy object allows to implement further subscript operators and delayed gather and scatter invocations to determine the SIMD vector entry type from the assignment operator.

The **`SubscriptOperation`** (Listing 4.24) class needs three template parameters: Obviously the type of the memory pointer and the type of the index vector/array/list need to be parameters. The third template parameter is needed for efficient implementation of the subscript operators in **`SubscriptOperation`**. This will be explained below.

4.8.6 simple gather/scatter operators

The proxy type implements the conversion operator for gathers (line 14) and the assignment operator for scatters (line 15). Both of these functions may only participate in overload resolution if the memory pointer type is an arithmetic type and the template parameter type **`V`** is a SIMD vector type.The gather and scatter operations are then, together with the template parameter type **`V`**, fully defined and thus support type conversion on load/store. The conversion is defined by the entry type of the SIMD vector type and the type of the memory pointer. At this point the number of entries in the SIMD vector is known and therefore the size of the

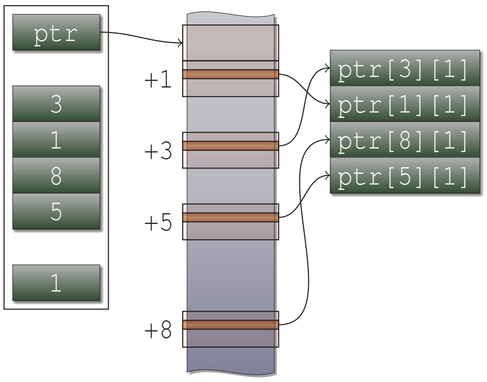


Figure 4.3: Graphical representation of the nested scalar subscript gather operation. (`ptr` is a nested array type, such as `array<array<float, 4>, 20>`)

index vector can be checked. If the size of the index vector is encoded in the type, then this will be used to additionally determine participation of the functions in overload resolution.[18]

Since scatter is implemented via an assignment operator in **`SubscriptOperation`**, it appears consistent to support compound assignment as well. In Vc I decided against doing so, because compound assignment implies that an implicit gather operation needs to be executed. Since gather/scatter are rather expensive operations, I believe the user should see more clearly that the memory access patterns of the code are sub-optimal. Not allowing compound assignment forces the user to explicitly execute a gather and a scatter, thus making the memory accesses more obvious.

The two operators implement unmasked gather and scatter. Masked gather scatter will be discussed in Section 5.4. The functions on lines 17 and 18 provide an interface to extract the necessary information in a minimal interface.

4.8.7 THE NESTED SCALAR SUBSCRIPT OPERATOR

The **`SubscriptOperation`** class (Listing 4.24) implements three subscript operators. The scalar subscript operator on line 24 allows to use gather/scatter for nested containers. The second operator on line 31 implements gather/scatter operations for different offsets in nested containers.[19] The third operator on line 37 enables gather/scatter to use arrays of structs.

18 An alternative to removing a function from overload resolution are diagnostics via `staticāssert` statements. (Compare footnote 11 on page 47.)

19 Basically, only nested arrays will work because of the requirement that only the outer container may allocate the data on the heap.

```
template <typename T,
          typename IndexVector,
          typename Scale = std::ratio<1, 1>>
class SubscriptOperation {
  const IndexVector m_indexes;
  T *const m_address;

  using ScaledIndexes = implementation_defined;

public:
  constexpr SubscriptOperation(T *address,
                               const IndexVector &indexes);

  template <typename V> operator V() const;
  template <typename V> SubscriptOperation &operator=(const V &rhs);

  GatherArguments<T, ScaledIndexes> gatherArguments() const;
  ScatterArguments<T, ScaledIndexes> scatterArguments() const;

  SubscriptOperation<
      std::remove_reference_t<decltype(m_address[0][0])>, IndexVector,
      std::ratio_multiply<
          Scale, std::ratio<sizeof(T), sizeof(m_address[0][0])>>>
      operator[](std::size_t index);

  template <typename IT>
  SubscriptOperation<
      std::remove_reference_t<
          decltype(m_address[0][std::declval<const IT &>()[0]])>,
      ScaledIndexes>
      operator[](const IT &index);

  template <typename U>
  SubscriptOperation<
      std::remove_reference_t<U>, IndexVector,
      std::ratio_multiply<Scale, std::ratio<sizeof(T), sizeof(U)>>>
      operator[](U T::*member);
};
```

Listing 4.24: The SubscriptOperation proxy type. It implements conversion to/from SIMD vector types via gather/scatter calls and additional subscript operators.

The scalar subscript operator (line 24) executes a constant array subscript for a nested array (see Figure 4.3). Semantically, the operator implements the same behavior as scalar subscripting. Thus, the expression `data[int_v::IndexesFromZero()][3]` references the elements `data[0][3], data[1][3], data[2][3],` … .

4.8.7.1 RETURN TYPE

The subscript operator must determine the new template arguments for the **`SubscriptOperation`** return type:

- The memory pointer previously pointed to an array of arrays. The new pointer must point to the beginning of the first array in the outer array. Thus, the type changes from array of **`U`** to **`U`**.

- The **`IndexVector`** type does not change at this point, also because its value is not modified.

- Because of the above, the index vector would now contain incorrect offsets. Consider the expression `data[int_v::IndexesFromZero()][3]` (as above) and assume `data` is of type `Vc::array<Vc::array<float, 100>, 100>`. Then `data[int_v::IndexesFromZero()]` returns an object of type `SubscriptOperation<Vc::array<float, 100>, int_v, ratio<1, 1>>`. The subsequent call to the nested scalar subscript operator (`operator[](std::size_t)`) determines the memory pointer type to be **`float`** and retains the index vector as `{0, 1, 2, …}`. Since `&data[1]` − `&data[0]` $= \frac{\mathcal{S}_{\texttt{Vc::array<float,100>}}}{\mathcal{S}_{\texttt{float}}} = 100$, the correct offsets to the new **`float`**-pointer are `{0, 100, 200, …}`. The pointer difference expression (`&data[1] - &data[0]`) is not a constant expression, but the `sizeof` fraction obviously is. Therefore, the `std::ratio` template argument is scaled with these two `sizeof` values (line 22).

  By using a template parameter, this fraction is built up in subsequent subscript operator calls and the division is evaluated at compile time inside the cast and assignment operators or the `gatherArguments` and `scatterArguments` functions. Thus, the multiplication of the index vector is delayed as far as possible. This is not only an optimization. It is necessary to delay the division to implement the member subscript operator (Section 4.8.10) correctly.

```
template <typename U = T>
auto operator[](enable_if_t<(has_no_allocated_data<T>::value &&
                             has_subscript_operator<T>::value &&
                             is_same<T, U>::value),
                            size_t> index)
    -> SubscriptOperation<
        remove_reference_t<decltype(m_address[0][index])>,
        IndexVector,
        ratio_multiply<
            Scale, ratio<sizeof(T), sizeof(m_address[0][index])>>>;
```

Listing 4.25: The complete declaration of the nested scalar subscript operator as used in Vc.

4.8.7.2 PARTICIPATION IN OVERLOAD RESOLUTION

The scalar subscript operator (line 24) may not be instantiated with the `SubscriptOperation<T, I, S>` class if `T` does not implement the subscript operator for arguments of `std::size_t`. This can be implemented via a dummy template parameter (`U`) to the subscript operator and modifying the subscript expression in the `decltype` expression such that it becomes a dependent type on `U`, while at the same time requiring `U` to be equal to `T`. In addition to delayed instantiation, the operator shall not participate in overload resolution if `T` does not implement the subscript operator or if `T` is a container that does not store its data inside the object. This last requirement is important to make the offset calculation work as described above. See Listing 4.25 for a possible declaration.

4.8.7.3 NESTED CONTAINER REQUIREMENTS

A container class that can be dynamically resized typically stores its data in a heap-allocated array outside of the container object. The object itself typically has only two or three member variables: pointers to begin and end of the data, and possibly a pointer to the end of the allocated memory. A well known example is `std::vector`. Consider what `Vc::array<std::vector<int>, N>` implies for SIMD gather and scatter operations, which require a single pointer to memory and a list of offsets to that pointer. In general, there is no guarantee about the dynamically allocated memory, thus the pointers of the `std::vector` objects could possibly cover the whole range of addressable memory.[20] On a system with a pointer-size of 64 bits, the index vector would be required to use 64-bit integers or risk pointing to incorrect addresses. Especially for SIMD vector gathers and scatters this is a very limiting requirement. The gather and scatter instructions in the MIC instruction set and the AVX2 gather instructions only support 32-bit integer offsets. In addition, these instructions assume unsigned integers and thus only positive offsets. The base pointer would therefore have to be the numerically smallest one.

20 or worse: different segments of memory

Overall, these constraints and complications show that an interface to gather/scatter for these kinds of nested container types is not worth the effort. The user would be much better served with writing a loop that assigns the scalar values sequentially.[21]

4.8.8 TYPE OF THE INDEX VECTOR

As mentioned above (and more below), at some point the index vector values must be scaled with the ratio stored in the `std::ratio` template parameter. However, when the multiplication is executed, the index vector type must be able to store these larger values. It would be easiest if the scaled index vector type were equal to the type used for the initial subscript. However, the scale operation of the indexes is implicit and not at all obvious to the user, who did not study the implementation of nested subscripting. The user only sees the need for the type of the subscript argument to be able to store the offsets for the outer subscript. Thus, a vector of 16-bit integers may appear to be sufficient.

Obviously, a 16-bit integer can quickly overflow with the scale operations involved for nested arrays.[22] Therefore, the index vector type needs to be promoted transparently before applying the scale operation. The safest type for promotion would be a `std::size_t`. However, as discussed above, the interface should preferably avoid 64-bit offsets, since they cannot be used for gather/scatter instructions on at least one major platform. Thus, integral promotion to `int` or `unsigned int` is the most sensible solution.

The promoted index vector type is captured in the `ScaledIndexes` member type (line 8). The discussion showed that there is no definite answer on the type promotion. Since the type is only used internally, the implementation may choose the exact rules.

For Vc I chose the following logic for the `ScaledIndexes` member type:

- If `IndexVector` is `Vector<T>` or `SimdArray<T, N>` and $\mathcal{S}_{\mathtt{T}} \geq \mathcal{S}_{\mathtt{int}}$, then `ScaledIndexes` is set to `IndexVector`.
- If `IndexVector` is `Vector<T>` or `SimdArray<T, N>` and $\mathcal{S}_{\mathtt{T}} < \mathcal{S}_{\mathtt{int}}$, then `ScaledIndexes` is set to `SimdArray<int, IndexVector::size()>`.
- If `IndexVector` is an array with known size (`std::array`, `Vc::array`, or fixed-size C-array), then `ScaledIndexes` is set to `SimdArray<promoted_type<T>, N>` (where `T` is the value type of the array).

21 Or even better: The API limitation uncovers the flaw in the data structure and leads to a redesign and better data structures.

22 Consider `Vc::array<Vc::array<float, 1000>, 1000> data`: The first subscript operator only works with values from 0–999, which easily fit into a 16-bit integer. However, with the second subscript those indexes must be scaled by 1000, thus exceeding the representable range.

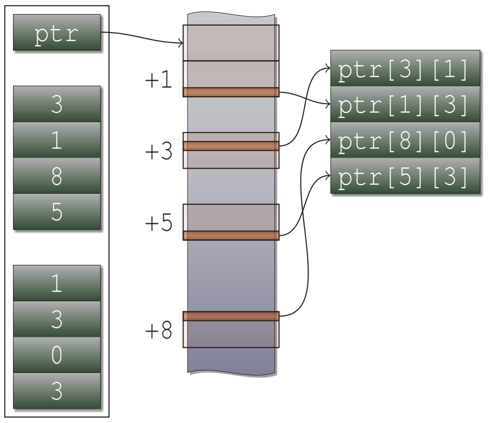


Figure 4.4: Graphical representation of the nested vector subscript gather operation. (`ptr` is a nested array type, such as `array<array<float, 4>, 20>`)

- If `IndexVector` is an `initializer_list<T>`, then `ScaledIndexes` is set to `vector<promoted_type<T>>`.
- If `IndexVector` is a `vector<T>`, then `ScaledIndexes` is set to `vector<promoted_type<T>>`.

4.8.9 THE NESTED VECTOR SUBSCRIPT OPERATOR

The second subscript operator on line 31 also implements gather and scatter on nested arrays (see Figure 4.4). In contrast to the above subscript operator, it allows to use different offsets on all nesting levels. Thus, `data[i][j]` references the values `{data[i[0]][j[0]], data[i[1]][j[1]], …}`. The second subscript's offsets therefore need to be added to the scaled original offsets. This is why the return type of the subscript operator is `SubscriptOperation<U, ScaledIndexes, ratio<1, 1>>`.

As for the scalar subscript operator, the vector subscript operator may not be instantiated together with the `SubscriptOperation<T, I, S>` class unless `T` implements the scalar subscript operator. Since the subscript operator is declared as a template function and the subscript expression in the deduction of type `U` of the return-type `SubscriptOperation<U, ScaledIndexes>` depends on the template type, this requirement is already fulfilled. In addition, the operator may only participate in overload resolution if and only if …

… `T` implements the subscript operator.

… `T` is a container that stores its data inside the object. (cf. Section 4.8.7.3)

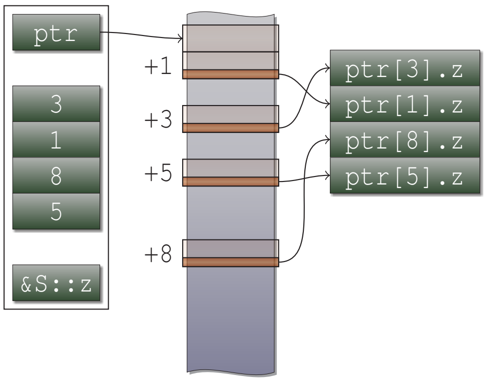


Figure 4.5: Graphical representation of the nested struct subscript gather operation. (`ptr` is an array of struct type, such as **`array`**`<S, 20>`, with `struct` **`S`** `{` **`float`** `x, y, z; };`)

… the operators function parameter type implements the subscript operator. This requirement fully disambiguates the function with the scalar subscript operator.

… the number of values in **`IndexVector`** and the function parameter type **`IT`** are equal or at least one of them cannot be determined as constant expression.

4.8.10 THE NESTED STRUCT MEMBER SUBSCRIPT OPERATOR

The third subscript operator on line 37 in Listing 4.24 enables gather and scatter for arrays of structs (see Figure 4.5). The API defined here does not have a direct counterpart in scalar code. The reason for this is that the dot-operator is not overloadable in C++ (yet).

Consider a container of type `Vc::`**`vector`**`<S>` with a simple structure (**`struct`** **`S`** `{` **`float`** `x, y, z; }`) for the following discussion. Then, to access **`S`**`::x` of a given array element with offset `i`, the required scalar code is `data[i].x`. The data member access via `.x` is not overloadable in a generic proxy-class returned from `data[indexes]` (where `indexes` is a vector/array of indexes). Therefore, with C++14, the only way to implement a vectorized struct gather requires `data[indexes]` to return a struct of proxy objects that are named `x`, `y`, and `z`. It follows that such an implementation must know the members of the struct before template instantiation. Such a return type therefore is not generically implementable with C++14. There is research going into the compile-time reflection capabilities required for creating such code [73].

4.8.10.1 EMULATING MEMBER ACCESS

Member access is too important to postpone its use with Vc until some future C++ standard provides the capabilities, though. Therefore, consider an alternative that, at least when seen in code, is suggestive enough to a developer that (s)he can understand intuitively what it does and how it can be used in new code.

The solution I chose in Vc therefore had to rely on a different operator overload or member function. The subscript operator is the semantically closest relative to accessing member variables via the dot operator. However, instead of an integral argument, the subscript operator needs to know the member variable offset in the struct, which can be expressed with pointers to members. Thus, `data[indexes][&S::x]` in Vc expresses the equivalent of `data[index].x` in scalar code.

4.8.10.2 PARTICIPATION IN OVERLOAD RESOLUTION

The nested struct member subscript operator may only participate in overload resolution if the template parameter **`T`** of the **`SubscriptOperation`** class template is a `class` or `union` type. Otherwise, the pointer to member type in the function parameter list would be an ill-formed expression.

4.8.10.3 RETURN TYPE

The return type of `operator[](U S::*)` follows the same considerations as the return type for `operator[](std::size_t)` (cf. Section 4.8.7.1). However, now the importance of using a fraction template parameter instead of immediate scaling or a single integer for the scaling parameter becomes clear. Consider a struct that contains an array: `struct S2 { float x[4], y; }`. The index scale factor for `&S2::x` thus would be $\frac{\mathcal{S}_{\texttt{S2}}}{\mathcal{S}_{\texttt{float[4]}}} = \frac{20}{16}$, which is not an integral value. In order to access a scalar element, the user must call another subscript operator for the `S2::x` array, which will result in the scaling fraction $\frac{\mathcal{S}_{\texttt{float[4]}}}{\mathcal{S}_{\texttt{float}}} = \frac{16}{4}$. The final scaling that is applied in e.g. `SubscriptOperation::operator V()` thus becomes $\frac{20 \cdot 16}{16 \cdot 4} = 5$.

4.8.10.4 DIGRESSION: IMPLICATIONS FOR `OPERATOR.()`

The **`SubscriptOperation`** class is a use case for an overloadable `operator.()` that needs to do more than just forward to an object returned by the operator (i.e. the current `operator->()` cannot be used either). A possible `operator.()` declaration for **`SubscriptOperation`** could look like the one in Listing 4.26. For a container of type `Vc::vector<S>`, the code `data[indexes].x` would thus call `data[indexes].operator.(&S::x)` The type **`S`** could be deduced from the parameter type of the `operator.()` declaration.

```
template <typename U>
SubscriptOperation<
    std::remove_reference_t<U>, IndexVector,
    std::ratio_multiply<Scale, std::ratio<sizeof(S), sizeof(U)>>>
    operator.(U T::*member);
```

Listing 4.26: A possible `operator.()` overload that can capture the requirements of **`SubscriptOperation`**.

```
namespace Vc {
using target_dependent::Vector;
using target_dependent:: float_v;
using target_dependent::double_v;
using target_dependent::   int_v;
// ...
}
```

Listing 4.27: Declaration of the default SIMD vector type(s).

4.9 SELECTING THE DEFAULT SIMD VECTOR TYPE

Section 4.1 (Listing 4.1) showed that the **`Vector`**`<T>` class is defined inside a *target-dependent* namespace. Thus, the class (and its type aliases) need to be imported into a public namespace. The `Vc` namespace therefore must import all vector types from one of the target-dependent namespaces with `using` declarations (Listing 4.27). The default choice is very important to the idea of a portable SIMD vector type because it allows using a different target-dependent implementation of **`Vector`**`<T>` with just a recompilation (using a different compiler or compiler flags).[23]

4.9.1 THE SCALAR VECTOR<T> IMPLEMENTATION

In addition to the vector types in the `Vc` namespace, the `Vc::Scalar` namespace is always defined. The `Vc::Scalar::`**`Vector`**`<T>` class is implemented with **`T`** as **`VectorType`** member type, thus storing a single scalar value. However, in contrast to the fundamental type **`T`**, `Vc::Scalar::`**`Vector`**`<T>` implements the complete SIMD types interface and thus is always available as a drop-in replacement for `Vc::`**`Vector`**`<`**`T`**`>`.

The scalar implementation is especially useful for

- making debugging a vectorized algorithm easier.
- testing that a given code works with a different vector width.
- targets without SIMD registers/instructions.

23 Since this has consequences on ABI compatibility, the default might need to be a little more conservative and user-configurable. Chapter 8 discusses the issue.

- implementing generic algorithms that need to be able to process chunks of data that are smaller than the native SIMD width (cf. Section 9.1).

4.9.2 SEVERAL SIMD GENERATIONS IN ONE TRANSLATION UNIT

For some target architectures it is possible to support more than one SIMD register width. This can be supported by using documented namespace names for the different SIMD targets. Then a user that knows that (s)he targets this architecture can explicitly use SIMD registers and operations that are not declared as the default type.

As an example consider a x86 target with AVX instruction support. In addition to 256-bit SIMD registers the target also supports 128-bit SIMD registers (SSE). Thus, the types `Vc::AVX::float_v` and `Vc::SSE::float_v` as well as `Vc::Scalar::float_v` are available to the user. (The type `Vc::float_v` would then be an alias for `Vc::AVX::float_v`.)

For a problem where only 4 entries in a single-precision float SIMD vector are needed `Vc::SSE::float_v` will perform better than `Vc::AVX::float_v` because it requires half of the memory bandwidth and because not all instructions on the CPU work equally fast for AVX vectors as for SSE vectors. As we will see later (Chapter 7), the different vector types can be abstracted into a higher level type, alleviating the need for `#ifdef`s checking for a specific target architecture.

Experience has shown that it is useful to forward declare all user-visible types from target-specific namespaces even in case they are incompatible with the target system of the current compilation unit. This makes it easier for users to write target-specific code without using the preprocessor, only relying on template instantiation rules.

4.9.3 IMPLEMENTATION CONSIDERATIONS

If you consider that there are definitions for all possible SIMD targets (and thus platform specific code) in different namespaces, the question remains how the library decides what namespaces are made visible and which one of them to choose as the default. This selection can (and must) be implemented with the preprocessor. This is obvious for hiding implementations that would be ill-formed because of the use of platform- and implementation-specific types. Choosing the default SIMD vector type could be possible with template programming, but still requires some constant expression that identifies the supported SIMD targets. Today, this information is not accessible through standard C++ and must rely on compiler-specific macros, making an implementation with preprocessor expressions straightforward.

For most target architectures there is a very limited set of choices with regard to SIMD. Today, one of the most diverse SIMD targets is the x86_64 architecture (an overview was given in Section 1.4). Let us consider a compilation unit with compiler flags set for the AMD Piledriver microarchitecture.[24] This microarchitecture supports the following SIMD (or SIMD related) instruction extensions: MMX, SSE, SSE2, SSE3, SSE4A, SSSE3, SSE4, F16C, FMA, AVX, XOP. Since the user will expect that in this case the most advanced registers and instructions will be used, the default **`Vector<T>`** implementation should use vector registers of 256 bits width for floating-point types and 128 bits for integral types.[25] See Chapter 8 for a discussion of the ABI implications.

4.10 INTERFACE TO INTERNAL DATA

The `Vc::`**`Vector`** class does not implement all operations that a user might want to use. Most importantly, there exist specialized instructions for specific application domains that are not easy to capture in a generic interface. Thus, if the Vc types should be usable also for such special application domains it must be possible to access internal and implementation-dependent data.

The `Vc::`**`Vector`** class therefore defines the `data()` member function and a constructor that converts from a **`VectorType`** object:

```
  Vector(VectorType);
  VectorType &data();
  const VectorType &data() const;
```

For example, a user might want to use the SSE intrinsic `_mm_adds_epi16`[26] with `Vc::SSE::`**`short_v`** and can thus write his/her own abstraction:

```
Vc::SSE::short_v add_saturating(Vc::SSE::short_v a,
                                Vc::SSE::short_v b) {
  return _mm_adds_epi16(a.data(), b.data());
}
```

The choice of the `data()` function is rather arbitrary. Alternatively, the **`VectorType`** could also be returned via a conversion operator (`operator` **`VectorType`** `&()`) or a non-member friend function such as (**`VectorType`** `&internal_data(`**`Vector<T>`** `&)`). The conversion operator might be the most convenient solution, though it can certainly be argued that it is too convenient because the implicit conversion can be unintended and nonobvious. However, the conversion operator could also be declared as `explicit`, in which case the `data()` member function is more convenient again.

24 For instance, the compiler flag for GCC for the AMD Piledriver microarchitecture is `-march=bdver2`.

25 Vc used 256 bits (two 128-bit operations per vector) for **`int_v`** and **`uint_v`** before Vc 1.0.

26 This intrinsic adds 16-bit integers with signed saturation, thus returning `SHRT_MIN`/`SHRT_MAX` if the addition would underflow/overflow. This functionality may certainly be abstracted for the `Vc::`**`Vector`** types, but currently this is not the case.

4.11 CONCLUSION

This chapter has introduced the API for a type that allows data-parallel execution for arithmetic types. On the one hand, I have shown how the interface has to be designed to follow the syntax and semantics of existing C++ as much as possible. On the other hand, I have shown where the semantics have to differ from the scalar types.

The resulting programming interface enables users to write explicitly data-parallel code without sacrificing serial semantics. Thus, the user does not have to reason about concurrent execution of serially stated algorithms but can reason about serially executing operations of which each processes data in parallel. Vc's `Vector<T>` type thus enables a data-parallel programming model which is easy to reason about without abandoning functionality.

5 DATA-PARALLEL CONDITIONALS

The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer.

— Edsger W. Dijkstra (1972)

Conditional statements are some of the most important language elements in C++. `if` statements enable programs to follow different code paths depending on arbitrary boolean conditions. In most cases an `if` statement is translated as a branching instruction. These instructions can be expensive on modern processors, if the branch prediction unit chooses the wrong branch. In such a case the pipeline has to be flushed and execution must restart at the other branch. This can incur penalties on the order of 100 cycles.

In order to overcome costly pipeline flushes on incorrect branch prediction, conditional move instructions have been introduced. A conditional move instruction typically executes a load or register copy if one or more specific flag(s) is/are set. Thus, an optimizing compiler might translate the code `if (condition) { x = y; }` into a compare instruction and a subsequent conditional move instruction.

Not every conditional jump in machine code is translated from an `if` statement. Conditional jumps are used for loop exit conditions in `while` or `for` statements. Furthermore, `switch` statements describe jumps into one out of several code sections, where each one can be identified via one or more integral value(s). Instead of a `switch` statement, the logic can alternatively be expressed as several `if` statements. This is functionally equivalent, but often compilers optimize `switch` statements via jump tables, while `if` cascades are typically translated as consecutive compares and jumps.

```
1 int_v f(int_v a, int_v b) {
2   if (a < b) {
3     a += 1;
4   } else {
5     a -= 1;
6   }
7   return a;
8 }
```

Listing 5.1: Example code relying on overloaded semantics for `if` statements with mask arguments.

5.1 CONDITIONALS IN SIMD CONTEXT

The SIMD types, as defined in Chapter 4 do not return booleans from the compare operators. Instead, they return `Vector<T>::MaskType`, which is an alias for `Mask<T>`. This mask type is the equivalent of a `Vector<bool>` type, but with additional type information about the associated `Vector<T>::EntryType`. (The need for this additional type information will be discussed in Section 5.2.) Thus, operations returning a definitive `true` or `false` answer with scalar types, return multiple `true` and/or `false` values in one return value with SIMD types. Obviously, these mask types cannot work directly with the builtin conditional statements in C++.

For SIMD code there are two principal choices for the semantics of `if`, `for`, `while`, and `switch`:

1. By enhancing the language and modifying compilers accordingly, it is possible to overload the meaning of conditional statements with operands of mask type. This has been implemented in Cilk Plus for the array notation extension [80]. Conditional statements subsequently do not disable a branch unless all entries of the mask are `false` (though essentially this is an optional optimization). Instead, all code branches are executed, only with some vector lanes implicitly disabled. Consider the example code in Listing 5.1 on a system with $\mathcal{W}_{\texttt{int}} = 4$ and `a = {1, 2, 3, 4}, b = {7, 0, 7, 7}`: The expression `a < b` then returns a mask with 4 boolean values: `{true, false, true, true}`. The compiler therefore has to translate the `if`-branch (line 3) into instructions that modify `a` only at the indexes 0, 2, and 3. Subsequently, `a` will be `a = {2, 2, 4, 5}`. The `else`-branch (line 5) then may only modify the SIMD vector entry at index 1. Thus, `a` must become `a = {2, 1, 4, 5}`, which is the return value of the function `f`.

2. The alternative keeps the semantics of the existing conditional statements unchanged. Then, mask types can only be used for conditional statements if a reduction function from a mask to a single boolean value is used (cf. Section 5.2.7). Still, the functionality described above (modifying a subset of

```
int_v f(int_v a, int_v b) {
  if (a < b) {
    return a + 1;
  } else {
    return a - 1;
  }
}
```

Listing 5.2: Code example that shows unclear return semantics: both branches must execute but from where does the function return and what is the return value?

```
int f(int_v a, int_v b) {
  if (a < b) {
    return +1;
  } else {
    return -1;
  }
}
```

Listing 5.3: Code example that shows unresolvable ambiguity: both branches must execute but there can be only one return value because the return type is a scalar **int**.

a SIMD vector, selected via a mask) can be implemented via write-masking expressions (cf. Section 5.3).

5.1.1 CONSEQUENCES OF IMPLICIT MASKING

Consider the implications of `if` statements that accept SIMD masks. The code example in Listing 5.2 is a small modification of the example in Listing 5.1 that would be equivalent for scalar types. However, with SIMD vector types both of the two `return` statements in the code must be taken. It is certainly possible to define that this code blends the SIMD vectors from the two `return` statements according to the implicit masks in the `if` and `else` branches. However, already a seemingly small change, such as returning an **int** instead of **int_v** (Listing 5.3) leads to unresolvable ambiguity: Should the function return `+1` or `-1`? Similar ambiguity issues occur with non-complementary masked `return` statements and function calls inside the branches. Throwing exceptions and locking/unlocking mutexes would even have to be disallowed altogether.

There is a more fundamental uncertainty resulting from implicit masking via `if` statements on SIMD vector masks: How should different SIMD vector types interact? An `if` statement from **int_v** comparison returns $\mathcal{W}_{\mathtt{int}}$ boolean answers. If the branch contains code with **short_v** or **double_v**, should it be implicitly write-masked or not? If yes, how? There is no natural and obvious behavior for applying write masks of different $\mathcal{W}_{\mathtt{T}}$.

This shows that `if` statements with non-boolean arguments limit the language features allowed in the `if/else` branches. This makes the feature much less intuitive. The implicit mask context changes the semantics significantly in different regions of the source code. And the problem is aggravated if a developer requires `else if` or `switch` statements.

5.1.2 DESIGN DECISION FOR VC

For the Vc library I therefore decided that the semantics of `if`, `for`, `while`, and `switch` must not change for explicit SIMD programming.[1] Everything else would be too surprising and unintuitive to users, especially developers that read existing code without prior knowledge about SIMD programming. This may sound obvious, but consider that many developers will start from a scalar implementation of their algorithm. In the scalar code the conditional statements correctly express the logic of the algorithm. When a developer subsequently vectorizes the code (s)he starts with replacing scalar types with the Vc vector types. At this point it may appear like a logical simplification of the vectorization process to keep the conditional statements unchanged in order to minimize the effort for the user. However, as discussed above, this comes at a considerable cost in consistency of semantics.[2] Thus, part of the issue is the question whether it is more important to ease initial vectorization of an algorithm or whether maintenance effort is more important. Even then, whether implicit write-masking via conditional statements eases initial vectorization at all certainly depends on the algorithm: The restricted semantics might lead to an even larger effort required for converting a given scalar code to SIMD code.

5.2 THE MASK<T> CLASS TEMPLATE

Analogous to the `Vector<T>` class discussed in Chapter 4, there needs to be a type acting as a SIMD vector of booleans. This is necessary for attaching the SIMD context to a type instead of implicit context. There are three main approaches:

- Reuse/Specialize the `Vector<T>` class (`Vector<bool>`).
- Define a new class (`Mask<T>`) with a type as template parameter.
- Define a new class (`Mask<Size>`) with a size as template parameter.

1 This is nice, because otherwise a pure library solution would not be possible.

2 There is not really a precedent in C++ for such differences in semantics / allowed operations for certain code regions. The transactional memory extensions for C++ [61] may introduce local semantics where actions inside a transaction are restricted to reversible operations. Approaches like explicit SIMD loops (Section 2.3) or the Intel Array Building Blocks framework [67] also rely on restricted local semantics.

5.2.1 WHY VECTOR<BOOL> IS NOT ENOUGH

The type **bool** is part of the *integral types* in C++. Since values of type **bool** "participate in integral promotions" [48, §3.9.1] they can be used in any expression where an **int** can be used.[3] Therefore, it appears as if the interface provided by **Vector**<**T**> is a good fit for boolean values, too. The additional functionality a SIMD vector of booleans should provide (such as the population count or reductions) could still be defined as non-member functions.

However, considering that $\mathcal{W}_\mathtt{T}$ may be different for different **T** it follows that $\mathcal{W}_\mathtt{bool} = \max(\{\mathcal{W}_\mathtt{T} \mid \forall\mathtt{T}\})$. Otherwise **Vector**<**bool**> would only be usable for a (target-dependent) subset of **Vector**<**T**> types. This definition of **Vector**<**bool**> implies that $\mathcal{W}_\mathtt{bool}$ may be greater than $\mathcal{W}_\mathtt{T}$ for some types **T**. Consider an SSE target, where $\mathcal{W}_\mathtt{short} = 8$, $\mathcal{W}_\mathtt{float} = 4$, and $\mathcal{W}_\mathtt{double} = 2$. Consequently, $\mathcal{W}_\mathtt{bool}$ would need to be 8 (16 if Vc::**Vector**<**signed char**> were supported by Vc) and store 50% or 75% unused data for masks interacting with **float_v** and **double_v**, respectively. Considering the implementation implications, this issue turns out to have serious efficiency implications, as well: With the SSE instruction set, boolean vectors are stored in the 128-bit SSE registers with 64/32/16/8 bits all set to either 0 or 1 for every associated value in the value vector. Thus, the hardware generates and expects booleans in different bit representations, depending on the SIMD vector type (or more accurately: $\mathcal{S}_\mathtt{T}$).

In addition to the size issue, there is good reason to use a single **bool** return value for the equal and not-equal comparison operators (cf. Section 5.2.6). Thus, **Vector**<**bool**> would need to specialize these functions, which is certainly possible, but, to a certain degree, defeats the purpose of using a generic class.

5.2.2 VC::MASK<T> DEFINITION

As discussed in Section 5.2.1, it is beneficial to define several mask types instead of a single boolean vector type. Looking at the SSE instruction set we have seen that **Mask**<Size> would suffice to define the minimal set of mask types for an x86/SSE target system. However, consider that the AVX instruction set uses $\mathcal{W}_\mathtt{float} = 8$ and $\mathcal{W}_\mathtt{double} = 4$ on top of the SSE vector sizes. Using the SIMD vector size as template parameter for the mask type thus would lead to subtle portability issues (this is the same issue discussed in Chapter 6 for **Vector**<**T**, N>): Consider the return types of the expressions **int_v**() == **int_v**() and **float_v**() == **float_v**(). With the SSE target they would both return the same **Mask**<4> type, whereas with AVX the types would differ: **Mask**<4> and **Mask**<8> respectively. The general solution (**Mask**<**T**>) therefore uses a different mask type for every SIMD vector type

3 "A prvalue of type bool can be converted to a prvalue of type int, with false becoming zero and true becoming one." [48, §4.5]

```
 1 namespace Vc {
 2 namespace target_dependent {
 3 template <typename T> class Mask
 4 {
 5   implementation_defined data;
 6
 7 public:
 8   typedef implementation_defined VectorType;
 9   typedef bool EntryType;
10   typedef implementation_defined EntryReference;
11
12   static constexpr size_t MemoryAlignment = implementation_defined;
13   static constexpr size_t size() { return implementation_defined; }
14   // ... (see below)
15 };
16 template <typename T> constexpr size_t Mask<T>::MemoryAlignment;
17
18 typedef Mask<         float>  float_m;
19 typedef Mask<        double> double_m;
20 typedef Mask<    signed int>    int_m;
21 typedef Mask<  unsigned int>   uint_m;
22 typedef Mask<  signed short>  short_m;
23 typedef Mask<unsigned short> ushort_m;
24 }  // namespace target_dependent
25 }  // namespace Vc
```

Listing 5.4: SIMD mask class definition.

`Vector<T>`.[4] That way the types are different for every target and the user will be forced to use explicit type conversions.

Listing 5.4 shows the definition of the SIMD mask type. Except for the `EntryType` member type all member types in Listing 5.4 are *implementation-defined*. This is analogous to the definition of the `Vector<T>` class in Listing 4.1. The different types are used for abstracting the following concepts:

`VectorType`
: (line 8) This type is used by the implementation to store a SIMD vector of booleans. For some implementations, this type may be equal to `Vector<T>::VectorType` but there is no such requirement.

`EntryType`
: (line 9) This is an alias for `bool`. The member type is defined for generality/interface compatibility with `Vector<T>`. This type signifies the conceptual entry type. The actual entries in `VectorType` may use a different binary representation than `bool`.

`EntryReference`
: (line 10) This type is used as the return type of the non-const subscript op-

4 Implicit and explicit conversions between `Mask<T>` and `Mask<U>` can be a no-op whenever $\mathcal{S}_T = \mathcal{S}_U \wedge \mathcal{W}_T = \mathcal{W}_U$.

```
Mask();
explicit Mask(bool);
template <typename U> Mask(const Mask<U> &);
template <typename U> explicit Mask(const Mask<U> &);
```

Listing 5.5: Declaration of the **Mask**<**T**> copy and cast constructors.

erator. It is therefore used to reference a single boolean entry in the internal mask representation. Note that the most compact binary representation for a SIMD vector of booleans uses a single bit per boolean value. In this case there cannot be a type representing the actual bits of the boolean value of a single mask entry.[5] Thus, **EntryReference** can also be a proxy class that can access (read and write) individual bits of such a mask via the assignment operators and cast-to-**bool** operator.

The **Mask**<**T**> type needs a single data member of an *implementation-defined* type (line 5). This member defines the size and alignment of the **Mask**<**T**> type.

The number of entries in the SIMD vector, in general, is different from $\mathcal{S}_{\texttt{Mask<T>}}$, which is why the `size()` constant (line 13) defines this value. To be consistent, the `size()` function must return the same value as **Vector**<**T**>::`size()`.

The **Mask**<**T**> type defines a `MemoryAlignment` static data member, just as **Vector**<**T**> does. Analogously, its value is the alignment requirement of pointers to **EntryType** (i. e. **bool**) in aligned load and store calls (Section 5.2.4). Implementation experience tells that in most cases the alignment of **Mask**<**T**> will not be equal to **Mask**<**T**>::`MemoryAlignment`. This is due the SIMD mask register either using several Bytes or only a single bit per boolean entry.

Finally, analogous to the type aliases for **Vector**<**T**>, the mask types implemented in the Vc library are aliased to the types **float_m**, **double_m**, … (lines 18–23).

5.2.3 CONSTRUCTORS

The constructors for the **Mask**<**T**> class need to replicate the semantics of the **bool** type as much as possible. The necessary declarations are shown in Listing 5.5.

The default constructor of **Mask**<**T**> initializes the value of all entries in the mask to `false`. This is required for compatibility with the expression **bool**`()`, which constructs a **bool** with the value `false`.

The copy and move constructors and operators are omitted for the same reason as for **Vector**<**T**> (cf. Chapter 4).

5 The object representation of any type in C++ takes up N bytes, where N is integral. This is also evident from the `sizeof` operator which returns a **size_t** denoting the number of bytes in the object representation of the type.

```
explicit Mask(const bool *mem);
template<typename Flags> explicit Mask(const bool *mem, Flags f);

void load(const bool *mem);
template<typename Flags> void load(const bool *mem, Flags);

void store(bool *) const;
template<typename Flags> void store(bool *mem, Flags) const;
```

Listing 5.6: Declaration of the **Mask**<**T**> load and store functions.

The constructor on line 2 initializes a mask object with all values set to the boolean value passed in the argument. Therefore, this constructor implements a broadcast from one scalar value to all entries in a SIMD vector. Note that, in contrast to the broadcast constructor from **Vector**<**T**>, the broadcast constructor of **Mask**<**T**> is declared as `explicit`. This is a deviation from the behavior of the scalar **bool** type. However, for boolean vectors the usefulness of a broadcast is mostly limited to initialization of mask objects. If a developer really needs to use a mask with all entries set to either `true` or `false`, then it is very likely that a scalar control-flow statement (such as `if`) is much better suited for the task. On the other hand, if implicit conversions from scalar **bool** to **Mask**<**T**> were possible, a user might fail to notice that an expression produces a **bool** instead of the intended mask object.

Finally, the two constructor functions on lines 3 and 4 implement implicit and explicit (`static_cast`) conversions between mask objects. The two functions, as declared in Listing 5.5, are ambiguous. They need to be adjusted, such that the implicit constructor only participates in overload resolution if $\mathcal{W}_U = \mathcal{W}_T$ for all possible targets. According to the discussion of implicit conversions in Chapter 4 this restriction can be implemented via the following `enable_if` expression:

```
enable_if<differs_only_in_signedness<U, T>::value>
```

The `explicit` constructor then simply requires the inverse condition to `enable_-if`.

5.2.4 LOADS AND STORES

Mask types can implement load and store functions, reading from / writing to arrays of **EntryType** (which is **bool**). These functions can be useful to write code that is independent of the SIMD register width and to interface with non-SIMD code (or I/O in general). Listing 5.6 shows the declaration of the necessary functions. The **Flags** argument is analogous to the one for the **Vector**<**T**> load/store functions. The default uses unaligned loads and stores and can be set to aligned load and store via the second argument.

```
Mask operator!() const;

Mask &operator&=(Mask);
Mask &operator|=(Mask);
Mask &operator^=(Mask);

Mask operator&(Mask) const;
Mask operator|(Mask) const;
Mask operator^(Mask) const;

Mask operator&&(Mask) const;
Mask operator||(Mask) const;
```

Listing 5.7: Declaration of logical and bitwise operators for **Mask**<T>.

```
  bool operator==(Mask rhs) const;
  bool operator!=(Mask rhs) const;
```

Listing 5.8: Declaration of the **Mask**<T> comparison operators.

5.2.5 LOGICAL AND BITWISE OPERATORS

Listing 5.7 shows the declaration of the operators for logical and bitwise operations. Each operator simply applies the operation component-wise. There is no need for non-member overloads as was required for **Vector**<T>, because the conversion rules are much simpler for different vectors of booleans. The implicit and explicit conversion constructors fully suffice.

5.2.6 COMPARISON OPERATORS

Listing 5.8 shows the declaration of the comparison operators that implemented for Vc::**Mask**. Note, that the return type is a scalar **bool** and not a SIMD type. Returning another mask type would make the compare operator basically an alias for the xor operator. Typically, it is more interesting to determine whether two given masks are equal (or not) and this requires a single boolean.

It is certainly possible to additionally define a meaning for relational compare operators (less/greater). The most obvious definition would be an interpretation of the boolean entries as bits of an integer and then compare the integers. Up to now I did not come across a use case for such operators, though, which is why I have not defined an interface for it.

5.2.7 REDUCTION FUNCTIONS

In order to use a mask object in an `if` statement or loop condition there needs to be a reduction function from the vector of boolean values in the mask to a single **`bool`**. The three reductions `all_of`, `any_of`, and `none_of` [48, §25.2.1–§25.2.3] present in the C++ standard library are applicable to masks. For Vc I added a fourth function:

`all_of`: Returns `true` if and only if all entries in the mask are `true`.

`any_of`: Returns `true` if and only if at least one entry in the mask is `true`.

`none_of`: Returns `true` if and only if all entries in the mask are `false`.

`some_of`: Returns `true` if and only if there is at least one entry that is `true` and at least one entry that is `false` (note that `some_of` always returns `false` for arguments of type `Vc::Scalar::Mask<T>`).

The usefulness of the first three functions should be obvious. The `some_of` reduction, however, is not used that often. It can be a useful check to determine whether a condition in the SIMD lanes diverged. For example, it could signify that a program still needs to continue iterating, but at least one vector lane is idle and a reorganization of the data vectors might increase the throughput.

The template functions reducing a mask object to a boolean need to be declared in such a way that they do not participate in overload resolution unless the template argument actually is a `Mask<T>` type (from any internal namespace).

In addition to the declarations for the `Vc::Mask` types, the reduction functions are also declared for **`bool`** arguments. That way the functions can be used in generic code where scalar types and `Vc::Vector` types can be used at the same time.

There are a few more useful reduction functions, which return integral values instead of a boolean. They are declared as member functions of `Mask<T>` (Listing 5.10):

POPULATION COUNT Returns how many values in the mask are `true`.

INDEX OF FIRST Returns the index of the first `true` value in the mask.

INDEX OF LAST Returns the index of the last `true` value in the mask.

BIT REPRESENTATION Returns the mask as one boolean entry per bit in an **`unsigned int`**.

```cpp
namespace Vc {
namespace target_dependent {
template <typename T> class Mask {
  // ...
  bool isFull() const;
  bool isNotEmpty() const;
  bool isEmpty() const;
  bool isMix() const;
  // ...
};
}  // namespace target_dependent

template <typename Mask> inline bool all_of(const Mask &m) {
  return m.isFull();
}
template <typename Mask> inline bool any_of(const Mask &m) {
  return m.isNotEmpty();
}
template <typename Mask> inline bool none_of(const Mask &m) {
  return m.isEmpty();
}
template <typename Mask> inline bool some_of(const Mask &m) {
  return m.isMix();
}

inline bool all_of(bool b) { return b; }
inline bool any_of(bool b) { return b; }
inline bool none_of(bool b) { return !b; }
inline bool some_of(bool) { return false; }

namespace target_dependent {
using Vc::all_of;
using Vc::any_of;
using Vc::none_of;
using Vc::some_of;
}  // namespace target_dependent
}  // namespace Vc
```

Listing 5.9: Declaration of the **Mask**<T> reduction functions.

```cpp
  int count() const;
  int indexOfFirst() const;
  int indexOfLast() const;
  unsigned int bits() const;
```

Listing 5.10: Mask reductions with integral return value.

```
std::valarray<int> data = {1, 6, 2, 5, 3, 4};
data[data > 4] = -1;
// now data == {1, -1, 2, -1, 3, 4}
```

Listing 5.11: `std::valarray` allows to assign to a subset of its entries via the subscript operator and a `std::mask_array` argument.

5.3 WRITE-MASKING

The term write-masking is used to denote the expression that disables an arbitrary set of vector lanes for writes to the destination register (or memory location). This is equivalent to the conditional move operation for scalars, applied to several values in parallel. Hardware support for write-masking requires a rather simple operation: instead of writing all bits from some temporary buffer to the destination register, some lanes are disabled, thus keeping the old value in the destination register unchanged.

From the language side, this operation has been implemented via implicit masking (such as the masked `if` statements in Cilk Plus [80]) or blend functions, which essentially implement the SIMD equivalent of the C++ ternary operator (conditional operator). However, `std::valarray` declares a write-masking syntax, which is very similar to the needs of the SIMD vector types (see Listing 5.11).

5.3.1 CONDITIONAL OPERATOR

For SIMD blend operations, the conditional operator (`a < b ? 1 : -1`) would be a very natural solution. It is straightforward to translate this conditional expression from scalar context into SIMD context. The operator expresses, that for a given condition, its result should be the value of either the first or the second expression after the question mark. In the SIMD case, where a boolean is replaced by a vector of booleans, the conditional operator states that the results of the first expression must be blended with the results of the second expression according to the mask in the conditional expression before the question mark.

With the current C++ standard, overloading the conditional operator is not allowed [48, §13.5]. (According to Stroustrup [75] "there is no fundamental reason to disallow overloading of `?:`", though.) Therefore, until C++ gains this ability, conditional operators have to be replaced by a function call for supporting SIMD types.[6]

For the Vc library, I defined the function

`Vector<T> iif(Mask<T>, Vector<T>, Vector<T>).`

The name `iif` is an abbreviation for *inline-if*. To allow generic use of this function,

6 Informal feedback from the C++ committee has been that there is interest for a concrete proposal on allowing overloads of the conditional operator.

```
void filter(/*...*/) {
  // ...
  float_t sigma2 = measurementModel.sigma2;
  float_t sigma216 = measurementModel.sigma216;
  float_t hch = measurementModel * F.slice<0, 2>();
  float_t denominator = Vc::iif (hch < sigma216, hch + sigma2, hch);
  // ...
}
```

Listing 5.12: Part of the Kalman-filter code that uses `iif`. The **`float_t`** type can be defined as either **`float`** or **`float_v`**.

```
WriteMaskedVector<T> operator()(MaskType);
```

Listing 5.13: Declaration of the function call operator for write-masking support in **`Vector`**`<`**`T`**`>`.

Vc provides the overload

`T` `iif(`**`bool`**`, ` **`T`**`, ` **`T`**`).`

Thus, `iif` can be used in template functions where both **`bool`**s and Vc mask types may be used as the first argument to `iif`. As an example, Listing 5.12 shows how `iif` is used inside the Kalman-filter. The **`float_t`** type can be defined to anything that returns either a boolean or a Vc mask on `operator<`. Thus, the implementation of the algorithm is generically usable for SIMD and scalar types.

5.3.2 WRITE-MASKED ASSIGNMENT OPERATORS

The `iif` function would suffice to translate any scalar conditional code to a vectorized code. However, it is neither a good general interface, nor does it properly express intention of the code, hiding behind unnecessarily complex expressions. Therefore, I created a new syntax for the **`Vector`**`<`**`T`**`>` types to express conditional assignment with any assignment operator:

```
x(x < 0) *= -1;
```

This line of code reads as: multiply `x` with `-1` where `x` is less than `0`. The general syntax is *vector-object* `(` *mask-object* `)` *assignment-operator initializer-clause*. The **`Vector`**`<`**`T`**`>` class template therefore declares the function call operator as shown in Listing 5.13. This operator returns a temporary object which stores a (non-const) lvalue-reference to the **`Vector`**`<`**`T`**`>` object and a copy of the mask object. The **`WriteMaskedVector`** class template overloads all assignment operators which implement the write-masked assignment to the **`Vector`**`<`**`T`**`>` object.

In addition to assignment operators the **`WriteMaskedVector`** can also implement the increment and decrement operators.

The choice for the function call operator (parenthesis) instead of the subscript operator (square brackets), which `std::`**`valarray`** uses, was deliberate. The semantics of the builtin subscript operator allow the expression to be assigned to and to be used as right-hand-side of an assignment. But **WriteMaskedVector** may only be used on the left-hand-side of an assignment. In addition, the expression `x[scalar < 0] *= -1` is valid C++, because **bool** is implicitly promoted to **int** and thus dereferences the entry either at offset `0` or `1`. It is very likely a coding error, though, which is why it is better not to reuse the subscript operator for write-masking. Furthermore, the subscript operator can be useful in a future extension of the **Vector**<**T**> interface to implement shuffles. Therefore, it is a clearer separation to attach the write-masking semantics to a different syntax.

5.3.2.1 ALTERNATIVE: Vc::WHERE

The function call operator syntax has a significant downside: It is impossible to write generic functions with conditional assignment that work with SIMD vector types and fundamental types. It would require an operator overload for fundamental types, or rather a change to the language specification. Therefore, I worked on alternative solutions:

```
Vc::where(x < 0, x) *= -1;   // variant (1)
Vc::where(x < 0) | x *= -1;  // variant (2)
Vc::where(x < 0) (x) *= -1;  // variant (3)
```

The goal was to have a function/expression that can return a `WriteMaskedVector` object for vector types and fundamental types.

- The first variant uses less “magic” but does not have such an obvious connection between the modified variable `x` and the assignment operator.

- The second variant states more clearly that an assignment to `x` is executed. However, it requires an operator between the `where` function and the assignee that has lower precedence than assignment operators. In any case, this operator will be deprived of its normal semantics, which is a potentially confusing solution.

- The third variant is a compromise of the first two variants. It uses the function call operator of the return type of the `where` function to make it clearer that assignment is applied to the `x` variable.

All three variants of the `where` function can be overloaded with fundamental types.

All four solutions for write-masking (`where` and **Vector**<**T**>`::operator()`) can be translated to optimal SIMD code and thus only differ in syntax. Vc will support the **Vector**<**T**>`::operator()` solution for backwards compatibility with previous versions in any case. However, the applicability of Vc in generic functions is so important that one of the `where` solutions must be available additionally. The members of the concurrency study group of the C++ committee expressed a preference for the first variant because the other variants use a foreign syntax. Basically, the argument is that the first variant is easiest to implement on the one hand and easiest to memorize for users on the other hand.[7]

5.3.2.2 RETURN TYPE OF MASKED ASSIGNMENT OPERATORS

The assignment operators that are declared in the `WriteMaskedVector` type can return either:

- A reference to the `Vector<T>` object that was modified.
- A temporary `Vector<T>` object that only contains the entries where the mask is `true`.
- The `WriteMaskedVector` object.
- Nothing (`void`).

Intuitively, the most sensible choice appears to be a reference to the modified `Vector<T>` object. However, then the statement `(x(x < 0) *= -1) += 2` may be surprising: it adds `2` to all vector entries, independent of the mask. Likewise, `y += (x(x < 0) *= -1)` has no obvious interpretation anymore because of the mask in the middle of the expression.

Consider that write-masked assignment is used as a replacement for `if`-statements. Using `void` as return type therefore is a more fitting choice because `if`-statements have no return value. By declaring the return type as `void` the above expressions become ill-formed, which seems to be the best solution for guiding users to write maintainable code and express intent clearly.

5.4 MASKED GATHER & SCATTER

Finally, let us look at masked gather and scatter operations. (Gather/scatter was introduced in Section 4.8.) A gather expression creates a temporary `Vector<T>` object that can be assigned to an lvalue. If the user wants to assign only a masked subset of the gathered values, the write-masked assignment as described in Section 5.3 suffices. However, write-masked gather is special in that there are memory reads which are unnecessary (and thus should be omitted for performance reasons) and potentially even invalid, out-of-bounds accesses. Therefore, we rather want write-masked assignment from a gather operation to propagate to the gather function itself. Then the gather function can use the mask to omit loads for the SIMD lanes that will not be used on assignment.

The scatter function, called from a scatter expression, must use the mask information for the same reasons: it should avoid unnecessary stores and must omit

7 Consequently, follow-up papers for data-parallel types to the C++ committee will focus on the `where` function as the sole solution.

out-of-bounds stores. However, for scatters the scatter expression is on the left hand side of the assignment operator and thus basically follows the same logic as normal write-masking. Therefore, masked scatters simply work as soon as the `SubscriptOperation` class supports the write-masking syntax introduced above.

To support masked gathers, the `WriteMaskedVector` class declares an assignment operator for an rvalue-reference to `SubscriptOperation`:

```
template <typename T, typename I, typename S>
void operator=(SubscriptOperation<T, I, S> &&);
```

The operator will call `gatherArguments` on the `SubscriptOperation` object and use that information to execute a masked gather and assign the result to the referenced `Vector<T>` object.

Note that this only allows direct assignment from the gather expression. The user can not execute additional operations (though this could be supported via expression templates).

5.5 CONCLUSION

In this chapter I have shown how an interface to conditionals in data-parallel algorithms needs to be defined. The `Mask<T>` type delivers a powerful interface to a vector of boolean values and can be used as a predicate to assignment.

The `Mask<T>` class and write-masking syntax enables users to write data-parallel code which clearly expresses what the computer will execute while still enabling data-dependent processing. Therefore, in contrast to loop-vectorization or extended `if` statements, no implicit write-masking and branch flattening is necessary. The Vc interface consequently makes bad algorithm or data structure design visible. It helps users understand the cost of conditionals and facilitates a search for optimizations.

6

VECTOR CLASS DESIGN ALTERNATIVES

There is no obvious natural choice for the class that provides the SIMD types. The choices I considered are:

1. Declare multiple `Vector<T>` class templates in separate namespaces for each possible SIMD target (Listing 6.1). This design has been discussed in Section 4.1 in detail and is the class design used in the Vc library.

2. Declare a single `Vector<T, SimdInterface>` class template (Listing 6.2).

3. Declare a single `Vector<T, Width>` class template (Listing 6.3).

6.1 DISCUSSION

At first glance the choices appear equivalent. It is possible to additionally provide the other two interfaces with any of the above choices. However, there are important differences: not every interface adaption can be done transparently. `Vector<T, SimdInterface>` can be declared as an alias template for `SimdNamespace::Vector<T>`. In this case the types are equal.

```cpp
namespace SSE {
  template <typename T> class Vector;
}
namespace AVX {
  template <typename T> class Vector;
}
namespace ...
#if DEFAULT_IS_SSE
using SSE::Vector;
#elif DEFAULT_IS_AVX
using AVX::Vector;
#elif
...
#endif
template <typename T, std::size_t N> class SimdArray;
```

Listing 6.1: The `Vector<T>` Design.

```
namespace Common {
  enum class SimdInterface {
    ...
  };
  template <typename T, SimdInterface I> class Vector;
}
template <typename T>
using Vector = Common::Vector<T,
#if DEFAULT_IS_SSE
                              Common::SimdInterface::SSE
#elif DEFAULT_IS_AVX
                              Common::SimdInterface::AVX
#elif
                              ...
#endif
                              >;
template <typename T, std::size_t N> class SimdArray;
```

Listing 6.2: The **Vector**<**T**, SimdInterface> Design.

```
template <typename T> constexpr std::size_t defaultWidth();
template <typename T, std::size_t N = defaultWidth<T>()> class Vector;
```

Listing 6.3: The **Vector**<**T**, Width> Design.

The **Vector**<**T**> types in the different namespaces in design 1 are different types. Consequently, they cannot be unified in a single class template **Vector**<**T**, SimdInterface>. This means, if a **Vector**<**T**, SimdInterface> type is desired, it has to be a new type and does not match objects of type **Vector**<**T**> (though providing implicit conversion is easy). The implication is that template argument deduction may be less convenient to use (see below).

It is easily possible to declare alias templates of **Vector**<**T**, Width> to **Vector**<**T**> in different namespaces. Equally, it can be aliased to or from **Vector**<**T**, SimdInterface> (see Listing 6.4). Nevertheless, **Vector**<**T**, Width> and **Vector**<**T**, SimdInterface> are not equivalent: Consider the type **Vector**<**float**, 8>. It is the same type irrespective of the target system supporting vectors of $\mathcal{W}_{\texttt{float}} = 8$ or not. Thus, a symbol (such as the function void f(**Vector**<**float**, 8>)) which is compiled for SSE and AVX would link, but crash or simply fail to work as expected at runtime. (This is due to **Vector**<**float**, 8> using two **__m128** member objects for SSE and only a single **__m256** for AVX.) The type **Vector**<**T**, Width>

```
template <typename T, std::size_t Width>
using Vector2 = Vector<T, simdInterfaceForWidth<T, Width>()>;

template <typename T, SimdInterface I>
using Vector2 = Vector<T, widthForSimdInterface<T, I>()>;
```

Listing 6.4: Possible aliasing of **Vector**<**T**, Width> to or from **Vector**<**T**, SimdInterface>.

```cpp
template <typename T> void f(Vector<T> v);
void g() {
  f(float_v());         // compiles
  f(double_v());        // compiles
  f(AVX::double_v());   // only compiles if AVX is the
                        // compile-time default anyway
}

// with Concepts-Lite:
template <SimdVector T> void f2(T v);
// alternative with C++14:
template <typename T> enable_if_t<is_simd_vector<T>::value> f2(T v);

void g2() {
  f2(SSE::float_v());
  f2(Scalar::double_v());
  f2(AVX::double_v());
}
```

Listing 6.5: Generic Function with Vector Parameter for design 1.

therefore does not ensure type safety and either needs a third template parameter to identify the SIMD architecture, or needs to be placed into a namespace as was done for `Vector<T>`. Therefore, the `Vector<T, Width>` type is a bad choice for the fundamental SIMD vector type. It is still a useful interface, though. Chapter 7 presents the details of a `Vector<T, Width>` class template.

In design 2, the fundamental SIMD vector class is more generic than in design 1. This is relevant for generic functions that are supposed to work for any possible SIMD vector type, independent of the current compile-time default. On the other hand, most code should only use the compile-time default SIMD vector type, therefore alleviating the issue. With design 1, a generic function that wants to accept any vector type, including from different namespaces, needs to use an unconstrained template parameter as function parameter type. In order to ensure that the generic function only works with a SIMD vector type parameter, `std::enable_if` can be used (see Listing 6.5). It is expected that a future C++ revision will introduce *concepts*, a solution that allows expressing requirements for template parameters [77].

With the current C++ standard, design 2 can express such a generic function more naturally (see Listing 6.6). If, on the other hand, a fully generic function wants to support both fundamental scalar types and SIMD vector types, `std::enable_if` or a concept are needed anyway.

```cpp
template <typename T, SimdInterface I> void f(Vector<T, I> v);
void g() {
  f(SSE::float_v());
  f(Scalar::double_v());
  f(AVX::double_v());
}
```

Listing 6.6: Generic Function with Vector Parameter for design 2 (design 3 is equivalent).

6.2 CONCLUSION

Design 3 is a bad choice for building the low-level type for writing data-parallel algorithms and data structures. Designs 1 and 2 appear like equally good design choices. Feedback from the C++ committee's study group on concurrency was that they would prefer to see design 2 realized for a vector type in the C++ standard. The reason why I chose design 1 for the Vc library is that it requires considerably fewer template constructions which work around restrictions of C++. The second template parameter in design 2 makes specialization of member functions for a given implementation impossible, because it requires partial template specializations for function templates, which is not supported in C++.

7

SIMDARRAY: VALARRAY AS IT SHOULD HAVE BEEN

The STL already defines a set of classes for data-parallel processing since its inception. This class has been used by some people for vectorization of their programs. There is very little encouragement in the C++ community to use the **valarray** classes, though.

7.1 STD::VALARRAY<T>

The C++ standard defines the **valarray** class as "a one-dimensional smart array, with elements numbered sequentially from zero" and as "a representation of the mathematical concept of an ordered set of values" [48, §26.6.2.1]. The `std::`**valarray**<**T**> class template thus can be used as a container which additionally implements all arithmetic, logical, bitwise, and compare operators for arithmetics manipulation. Application of such a binary operator evaluates the corresponding elements of the operands using the operator for type **T**. The order of element-wise evaluation is unspecified and therefore, it is undefined behavior if an operator call on one element depends on another element.

The standard allows implementors of the STL to implement **valarray** with expression templates [82], which can make sequences of operator calls cache-efficient, even though the expression is executed on the complete container.

The number of elements in **valarray**<**T**> is set at runtime with the call to the constructor or the member function `resize`. This interface therefore implies that **valarray**<**T**> must use some form of dynamic memory allocation. Additionally, all operations thus have to rely on runtime bounds.

The above has substantial consequences limiting the usefulness of **valarray**<**T**>:

```cpp
template <typename T>
valarray<T> f(const valarray<T> &a, const valarray<T> &b) {
    return a + b;
}
```

Listing 7.1: This function might exhibit undefined behavior.

- The `valarray<T>` interface is not fully type-safe. The function `f` in Listing 7.1 compiles fine. However, *ISO/IEC 14882:2011* [48, §26.6.3.1] states: "If the argument arrays do not have the same length, the behavior is undefined." It is obviously much easier for development if invalid operations are caught at compile time, which is only possible if the type encodes all information necessary to determine which operations are safe and which cannot work. For `valarray` this is not possible because the size is unknown at compile-time.

- `valarray<T>` cannot be implemented in a way that it generically supports the optimal SIMD implementation of the expressed algorithm. This is due to the fact that the type carries no information about the number of elements. As a consequence all `valarray<T>` functions must work for any number of elements.

- Using `valarray<T>` with an element count that matches the SIMD width of the target system neither guarantees vectorization nor can it create the optimal SIMD implementation, unless the container size were visible as a constant expression to the compiler.

- Even if operations with operands `T` and `U` are well-defined, the same operation is not defined for objects of types `valarray<T>` and `valarray<U>`.

As we have seen, the `valarray<T>` class expresses data-parallelism. However, it fails to express it in a way that makes it suitable for optimal vectorization and/or optimal memory cache usage (small working sets for better cache and TLB[1] usage). The next section describes an alternative that builds upon the SIMD vector types described in Chapter 4 and Chapter 5.

7.2 VC::SIMDARRAY<T, N>

The class template `SimdArray<T, N>` implements the same interface as `Vector<T>`. The only difference is in the internal data member(s) and thus also the interface to access the internal data. A template instantiation requests a certain number of elements in the data-parallel type, which needs to be mapped onto the hardware capabilities of the target system.

As an example consider the type `SimdArray<float, 8>` on x86_64: If the target system does not support AVX then the implementation must use two `SSE::Vector<float>` members. With AVX it can use a single `AVX::Vector<float>` member.

This has the same problem as discussed for `Vector<T, Width>` in Chapter 6. Therefore, the `SimdArray<T, N>` class needs a third template parameter to make

1 translation look aside buffer

```
template <typename T, std::size_t N,
          typename VectorType = select_best_vector_type<T, N>>
class alignas(nextPowerOfTwo((N + VectorType::size() - 1) /
                             VectorType::size()) *
              sizeof(VectorType)) SimdArray;
```

Listing 7.2: The **SimdArray**<**T**, N> class template definition.

```
typedef Vc::float_v float_v;
typedef Vc::SimdArray<double, float_v::size()> double_v;
typedef Vc::SimdArray<int, float_v::size()> int_v;
typedef Vc::SimdArray<unsigned int, float_v::size()> uint_v;
```

Listing 7.3: The declaration of SIMD types with an equal number of scalar entries enables simple conversion between different data types.

the SSE and AVX variants unique types. The class template therefore is declared as shown in Listing 7.2. For implementation purposes, it may be useful to add a fourth defaulted template parameter, which is initialized to `VectorType::size()`. That way the class can be specialized for `N == VectorType::size()`.

The `alignas` attribute is specified in order to ensure the same alignment on all target systems. That way, if the in-memory representation is equal, the memory for one SIMD implementation can be copied to code with a different SIMD implementation.

7.3 USES FOR SIMDARRAY

The `SimdArray<T, N>` class is not a drop-in replacement for `valarray`. The important difference is that `SimdArray<T, N>` is compile-time sized. Therefore, the working-set size must be derived from constraints in the algorithm or the involved data types.

The intended use of `SimdArray<T, N>` is with a value for `N` that is small. Thus, a typical value for `N` is somewhere between 2–16, possibly 32. Larger values create large working sets which can become less efficient to process for the CPU.

7.3.1 CONVERSION BETWEEN DIFFERENT DATA TYPES

The most important use for `SimdArray<T, N>` is the declaration of portable SIMD types that can easily convert between different underlying arithmetic types. This requires the SIMD types to have the same number of scalar entries. For example, with `SimdArray<T, N>` at hand, a developer can, if (s)he determines that his/her main data type in data structures and algorithms is `float`, use the type aliases shown in Listing 7.3. With these types, conversions between `float`, `double`, `int`, and `unsigned int` do not pad zeros or drop values. This solves the issue discussed in Section 4.2.2.

7.3.2 PORTABLE CODE FOR FIXED-WIDTH ALGORITHMS

Some problems can be expressed more clearly and efficiently with a fixed width of the vector type. For example, consider a problem where only three or four values are available for processing in parallel. In this case the developer can use `SimdArray<float, 4>` instead of `float_v`. This allows to optimize better in case a target supports different vector widths, such as x86_64. Instead of using `AVX::float_v`, `SimdArray<float, 4>` would then use `SSE::float_v`. This uses less memory, faster load/store instructions, and possibly faster arithmetic instructions. Compare Section 12.1 for an example for this use-case.

7.4 IMPLEMENTATION CONSIDERATIONS

7.4.1 EXPRESSION TEMPLATES

Since the `SimdArray<T, N>` class is possibly composed of multiple `Vector<T>` objects (depending on $N_V = \frac{\mathrm{N}}{W_{\mathrm{T}}}$), the use of expression templates needs to be considered. Expression templates can solve the problem that a sequence of operator calls is normally executed one operator after another. However, each operator call on `SimdArray<T, N>` must forward to N_V operator calls on the `Vector<T>` members. This leads to a proliferation of temporaries and thus increases register pressure (register values must be stored to memory before the next operator is executed). With expression templates the order of evaluation can be transposed, such that the inner iteration is over the operator calls and the outer iteration over the `Vector<T>` members.

There are reasons why expression templates are not needed or may even perform worse in this situation, though:

1. Because of instruction level parallelism (cf. Section 1.6.1), it may be more efficient to execute the sequence of operators in the unmodified order. Consider $N_V = 4$. Then `x * y * z` can execute in 11 cycles with a CPU where the multiplication instruction has a latency of 4 cycles and a throughput of 1 cycle (cf. [39]). Because the operator call `x * y` compiles to four multiplications of `Vector<T>`: `tmp0 = x0 * y0; tmp1 = x1 * y1; tmp2 = x2 * y2; tmp3 = x3 * y3;`.

2. Expression templates make the interface more complicated for the compiler and increase compile time. Additionally, the diagnostic message from the compiler on incorrect use can become very hard to parse for a human. If there is no clear evidence of improvement this cost rather should be avoided.

```cpp
template <typename T, std::size_t N, typename VectorType>
class SimdArray {
  static constexpr std::size_t N1 = nextPowerOfTwo(N - N / 2);
  static constexpr std::size_t N2 = N - N1;
  SimdArray<T, N1> data1;
  SimdArray<T, N2> data2;
};
```

Listing 7.4: With a recursive implementation of **SimdArray**<**T**, N>, arbitrary values of N are possible. This is most efficiently implemented with a first member that uses a power of two value for N and a second member that covers the remainder. Note that the third template argument is omitted to have select_best_vector_type determine the best type. Thus, **SimdArray**< **float**, 13> for an AVX target will be built from one AVX::**float_v**, one SSE::**float_v**, and one Scalar::**float_v**.

3. Since C++11, expression templates have the issue that assignment to a new variable declared with **auto** will store an expression object instead of the result of the evaluated expression. This can be surprising and makes the interface slightly fragile. (There is effort going into solving this issue for a future revision of the C++ standard.)

The intended use of **SimdArray**<**T**, N> is with a relatively small N argument (i.e. **SimdArray**<**T**, N> is not a container class). With small N, N_V will also be small, so that register pressure is typically in a manageable range for the CPU.

7.4.2 recursive versus linear implementation

The **SimdArray**<**T**, N> class can generally be implemented in two different ways:

1. A **SimdArray**<**T**, N> object stores $N_V = \frac{\mathtt{N}}{\mathcal{W}_{\mathtt{T}}}$ **Vector**<**T**> objects in an array.
2. A **SimdArray**<**T**, N> object stores two **SimdArray**<**T**, N / 2> objects unless N == **Vector**<**T**>::size(), in which case the **SimdArray**<**T**, N> object stores a single **Vector**<**T**> object.

The first implementation strategy requires the use of loops in order to implement the member functions of **SimdArray**<**T**, N> in a generic way. The second implementation strategy can be implemented as recursive function calls. This is especially useful for reduction functions where the reduction can thus easily be executed as a tree-like fold. Furthermore, the second strategy allows an implementation that allows any possible value for N as is shown in Listing 7.4.

For the Vc library I have used the second implementation strategy. This can be translated to perfect machine code by current C++ compilers. However, compile time for complex uses of **SimdArray**<**T**, N> is rather high.

7.5 MASK TYPES

Analogously to the `Mask<T>` class template a mask type is needed to correspond to `SimdArray<T, N>`. This `SimdMaskArray<T, N>` type implements the `Mask<T>` interface with the same implementation strategy as `SimdArray<T, N>`. Furthermore, the mask reduction functions (Section 5.2.7) need to be overloaded for `SimdMaskArray<T, N>`.

7.6 CONCLUSION

This chapter has presented class templates that solve the `Vector<T, N>` use-case in a slightly safer way. The `SimdArray<T, N>` and `SimdMaskArray<T, N>` types are built on top of the low-level `Vector<T>` and `Mask<T>` types and show that different efficient abstractions for data-parallel computing can be built on top of `Vector<T>`. The combination of `Vector<T>` and `SimdArray<T, N>` increase usability of the vector type interface (cf. Section 7.3.1) making several use-cases easier to express.

8 ABI CONSIDERATIONS

8.1 ABI INTRODUCTION

An *Application Binary Interface* (ABI) describes machine-, operating system-, and compiler-specific choices that are not covered by a programming language standard. The most important issues are function calling conventions, memory/stack organization, and symbol name mangling. For Linux the ABI is standardized and documented [64, 50]. For Windows the ABI is implicitly defined by its development tools. (Other compiler vendors either provide incompatible products or reverse-engineer the Windows ABI.) For all targets, the goal is to have an ABI that allows interoperability. Developers expect that their choice of compiler (and compiler version) does not have an influence on the TUs[1] that can be linked together correctly. Compiler vendors and operating system vendors have a great interest in providing this guarantee.

8.2 ABI RELEVANT DECISIONS IN VC

The interface choices for Vc have a direct influence on the ABI of the Vc library.

8.2.1 FUNCTION PARAMETERS

If a `Vector<T>` is used as a function parameter, there are two principal choices for implementing the parameter passing in the function call convention:

1. The vector can be passed as a single SIMD register.
2. The vector is pushed onto the stack and thus passed via memory.

Choice 1 is the most efficient choice for the majority of situations. This requires a trivial copy constructor and destructor in `Vector<T>` (which recursively requires the copy constructor and destructor to be trivial for all non-static data members) with the Linux ABI [64].

1 translation units

If a `union` of intrinsic type and array of scalar type is used to implement the data member of **`Vector`**`<T>`, the Linux x86_64 ABI requires different parameter passing, which is derived from all members of the `union`. With a trivial copy constructor and destructor a `union` with $\mathcal{S} = 16$ must be passed as two SSE registers or two general purpose registers, while with $\mathcal{S} = 32$ the parameter must be passed via memory[2] [64, §3.2.3]. (According to the rules in [64, §3.2.3] there is a workaround to achieve parameter passing via SIMD registers: The array inside the `union` must be declared with zero entries.[3])

This shows that, in addition to the interface definition, the concrete implementation strategy also has an influence on the resulting ABI of the vector types. This needs to be considered carefully when implementing the library. Consequently, an implementation should avoid a `union` based implementation and rather use a different compiler extension for explicit aliasing, such as GCC's `may_alias` attribute.

The discussion above equally applies to **`Mask`**`<T>` and all derived types, of course.

8.2.2 LINKING DIFFERENT TRANSLATION UNITS

A user can compile two TUs with different compiler flags for the target microarchitecture (for example, so that one is compiled for SSE and the other for AVX). This most likely happens with one TU in a library and the other in an application. Then Vc vector or mask types in the interfaces between the TUs are incompatible types. The most complicated architecture probably is x86: Very old systems have no usable SSE support, old systems support SSE, current systems AVX or AVX2, and future systems AVX-512. $\mathcal{W}_\mathrm{T}$ is different: `xmm` vs. `ymm` vs. `zmm` registers. Concluding, it might be a good idea that packagers do not treat x86 as a single architecture anymore, but several ones. There is great interest in not having to take that path, though. The following sections will explore the issue in detail.

8.3 PROBLEM

The **`Vector`**`<T>` type is defined as a target-dependent type, which, similarly to **`int`**[4], uses the most efficient register size on the target system. For SIMD registers this implies that the number of values $\mathcal{W}_\mathrm{T}$ stored in a **`Vector`**`<T>` object can be different between different microarchitectures of the same architecture. The SIMD Types interface (Chapter 4) at least ensures that the types **`Vector`**`<T>` are different

2 GCC implements it this way. The clang/LLVM compiler (version 3.4) passes `union`s with $\mathcal{S} = 32$ via a single AVX register, though.

3 Clang/LLVM (version 3.4) in this case inverts the behavior for AVX `union`s and passes via memory.

4 "Plain ints have the natural size suggested by the architecture of the execution environment" [48, §3.9.1 p2]

if the register sizes differ. Therefore, the use of `Vector<T>` is safeguarded against incompatible linking, which would result in spurious runtime errors.

For the following discussion, consider an Intel Haswell system, which implements the x86_64 architecture and AVX2 SIMD registers & operations as a part of its microarchitecture (for simplicity, ignore the MMX instruction set). Then,

- with AVX2 $\mathcal{W}_{\texttt{float}} = 8$ and $\mathcal{W}_{\texttt{int}} = 8$,
- with AVX $\mathcal{W}_{\texttt{float}} = 8$ and $\mathcal{W}_{\texttt{int}} = 4$,
- with SSE $\mathcal{W}_{\texttt{float}} = 4$ and $\mathcal{W}_{\texttt{int}} = 4$, and
- without using SIMD functionality $\mathcal{W}_{\texttt{float}} = 1$ and $\mathcal{W}_{\texttt{int}} = 1$ (the `Scalar` implementation mentioned in Section 4.9.1).

The `Vector<T>` incompatibility between different SIMD instruction sets implies that a TU built for Intel SandyBridge differs in ABI to a TU built for Haswell. This breaks with the guarantee compiler vendors would like to retain: the ABI for a given architecture should stay stable. With the current `Vector<T>` proposal, implemented on top of SIMD intrinsics, the ABI would only be stable within microarchitectures.

One could argue that it is technically correct that some microarchitectures (those with differing SIMD widths) of the same architecture are partially incompatible, and thus the ABI could/should reflect this. On the other hand, it is very desirable that such incompatibilities are either hidden from (or consciously enabled by) the user. Thus, if it is at all possible to have the compiler automatically adapt between the microarchitectural differences, then implementors should invest in getting the `Vector<T>` ABI right from the outset.

8.3.1 FIXED $\mathcal{W}_{t}$ IN INTERFACES IS NOT THE SOLUTION

A common idea for solving the above issue, is to request that the SIMD type uses a user-defined width (cf. Fog [25] and Wang et al. [83]). Then the type would use the same $\mathcal{W}_{\texttt{T}}$ on any target and the types would be equal in different TUs.

There are two issues with this:

1. There is no guarantee that the specific $\mathcal{W}_{\texttt{T}}$ can be implemented efficiently on all target systems. Consider, for example, the common choice of $\mathcal{W}_{\texttt{float}} = 4$ compiled for an Intel Xeon Phi. The type would have to be implemented with a 512-bit SIMD register where 75% of the values are masked off. On a target without SIMD support, four scalar registers would have to be used, which increases register pressure.[5]

5 With luck this might just be the right loop-unrolling to achieve good performance, but it is the wrong mechanism to achieve this effect.

2. Even though the types are equal, the specific parameter passing implementation might be different. Consider a `vec<float, 8>` type translated for either AVX or SSE. Then the function

   ```
   void f(vec<float, 8>)
   ```

   would use `ymm0` with AVX and `xmm0` and `xmm1` with SSE to pass the function parameter from the caller to the function. Thus, if this were the preferred solution for implementors, vector types would have to be passed via the stack for function parameter passing (cf. Section 8.2.1). In addition, the in-memory representation and alignment requirements for the different microarchitectures must be defined in such a way that they work correctly on all systems.

From my experience, and in order to enable full scaling to different SIMD targets, I prefer a solution where a fixed $\mathcal{W}_{\mathtt{T}}$ is only chosen because it is dictated by the algorithm, not because of technical complications with ABI compatibility.

8.3.2 DERIVED TYPES

A class that is derived from `Vector<T>` or a class that has a non-static `Vector<T>` member will not have a different type in different TUs which are compiled for different SIMD widths. Thus, the linkage safety built into `Vector<T>` does not work for any derived types. Furthermore, this suggests that a solution that transparently adapts the ABI differences must be rather invasive.

The compiler would have to compile Scalar, SSE, AVX, and AVX2 (to stay with the x86_64 example) variants of all derived types and functions that use these types. The symbols would need additional information about the SIMD target as part of the name mangling.

Automatic adaption (such as a call from an AVX TU to an SSE TU) between derived types will be a problem, though. Consider that TU1 creates an object of a derived type `D`. A call to a member function, which is not declared `inline` and instead was compiled for a different SIMD width in TU2 now would require a transparent conversion of the object from one SIMD width to a different SIMD width. There cannot be a generic strategy to perform such a conversion without breaking the semantics guaranteed to the implementation of `D`.

```
// a.cc (SSE2 : float_v::size() == 4):
static float_v globalData;
void f(float_v x) { globalData = x; }
float_v g() { return globalData; }

// b.cc (AVX2 : float_v::size() == 8):
float_v h(float_v x) {
  f(x);  // calls f(x[0...3]) and f(x[4...7])
  // now globalData is either x[0...3] or x[4...7], depending on the order of
  // calls to f above
  return g();  // calls concatenate(g(), g())
}

int main() {
  cout << h(float_v::IndexesFromZero());  // {0 1 2 3 4 5 6 7}
  return 0;
}

// prints:
// 0 1 2 3 0 1 2 3
// or:
// 4 5 6 7 4 5 6 7
```

Listing 8.1: Impure functions break the adaption strategy of using multiple calls to TUs with shorter SIMD width.

8.3.3 serial semantics

Consider an ABI adaption strategy that splits a function call from TU1 with a `Vector<T>` argument with $\mathcal{W}_{\mathrm{T}}^{(1)}$ to multiple function calls to the function compiled with $\mathcal{W}_{\mathrm{T}}^{(2)} = \frac{\mathcal{W}_{\mathrm{T}}^{(1)}}{N}$ in TU2. This approach exposes non-serial semantics. This manifests, for instance, if two functions are intended to be called in serial succession, communicating via a global (or thread-local) variable.[6] If the adaption from an AVX2 TU to an SSE TU is done via calling the SSE function twice with the low and high parts of the vector argument, then the first function will be called twice, before the second function is called twice.

Consider the example in Listing 8.1. The developer expected serial semantics in function `h`. Instead, `f` is called twice, before `g` is called twice. Therefore, the conclusion is that adapting between different SIMD widths cannot be done via splitting a function call into multiple function calls.

6 This is probably a bad design, but that does not invalidate the problem.

8.3.4 LARGEST COMMON SIMD WIDTH

Consider a compiler implementation that identifies types that depend on $\mathcal{W}_{\mathtt{T}}$ and automatically compiles these symbols for all possible $\mathcal{W}_{\mathtt{T}}$ the target supports (extending the mangling rules accordingly). Then, when the TUs are linked to a single executable, the linker can detect whether for some symbols some $\mathcal{W}_{\mathtt{T}}$ translations are missing. In this case it can drop these $\mathcal{W}_{\mathtt{T}}$ symbols. The same could be done by the loader when the program is dynamically linked, right before executing the program. The largest remaining $\mathcal{W}_{\mathtt{T}}$ symbols can then be used to execute the program.

This solution should work as long as no dynamically loaded libraries are used (e.g. Plug-ins). Because, if an incompatible library (i.e. one that does not have the symbols for the currently executing $\mathcal{W}_{\mathtt{T}}$) is loaded, the program cannot switch back down to a smaller $\mathcal{W}_{\mathtt{T}}$. Thus, at least the ABI compatibility with dynamically loaded symbols cannot be guaranteed by this approach.

8.3.5 SIMD-ENABLED FUNCTIONS

The SIMD-enabled functions described in [30] provide the semantic restriction which works around the issue described in Section 8.3.3. The code in Listing 8.1 would still produce the same result, but because of the semantic restriction for the functions `f` and `g` the undefined behavior would be expected.

On the other hand, a member function of a class with members of vector type that accesses such members will still not be automatically adaptable between different TUs. Consider Listing 8.2. The call to `D::f` on line 21 will pass a `this` pointer to an object storing two **`float_v`** objects with $\mathcal{W}_{\mathtt{float}} = 8$ placed next to each other in memory. The function `D::f`, on the other hand, (line 10) expects two **`float_v`** objects with $\mathcal{W}_{\mathtt{float}} = 4$ consecutively in memory (Figure 8.1). In order to adapt such differences between TUs automatically, the adaptor code would have to create two temporary objects of type **`D`** (with the ABI in `a.cc`), copy the data, call the function `D::f` twice, copy the resulting temporary objects back into the original object and return. But such a strategy breaks with the call to `next->f()`. Non-vector members cannot be transformed generically and the `next` pointer would therefore point to an untransformed object.

Effectively, the strength of vector types (namely target-optimized data structures) inhibits the creation of automatic ABI adaption between TUs with different $\mathcal{W}_{\mathtt{T}}$.

```cpp
typedef Vector<float, Target::Widest> float_v;
struct D {
  float_v x, y;
  unique_ptr<D> next;
  D() : x(float_v::IndexesFromZero()), y(0) {}
  void f() [[simd]];
};

// a.cc (widest float_v::size() == 4):
void D::f() [[simd]] {
  y = (y + 1) * x;
  if (next) {
    next->f();
  }
}

// b.cc (widest float_v::size() == 8):
int main() {
  D d;
  d.next.reset(new D);
  d.f();
}
```

Listing 8.2: Member functions as SIMD-enabled functions?

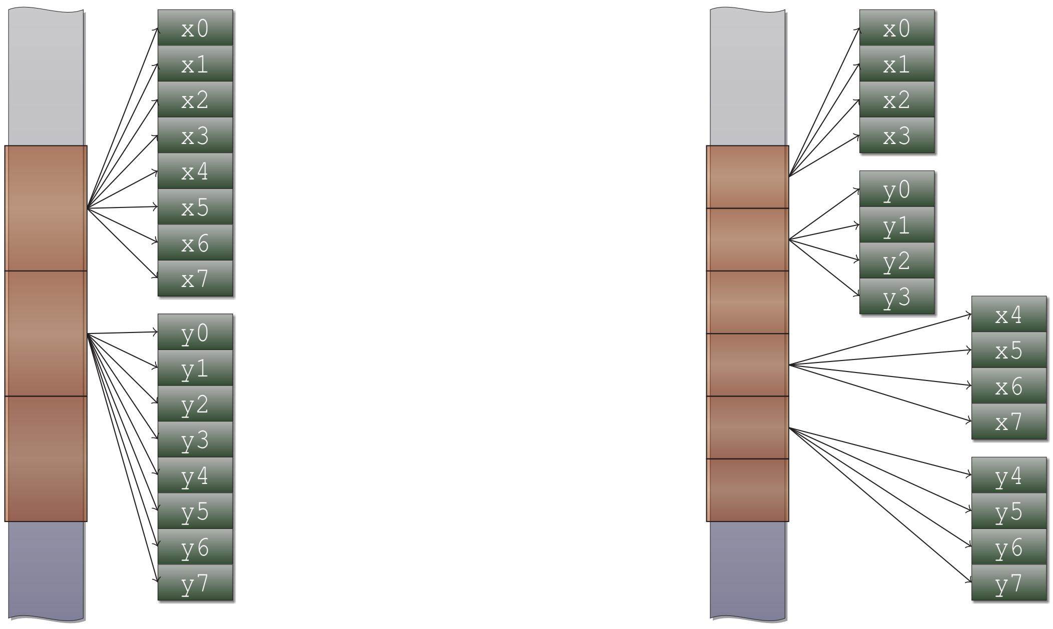


Figure 8.1: Memory layout differences depending on ABI. The memory layout of d in the caller is shown on the left. However, the function D::f expects the memory layout as shown on the right.

8.4 SOLUTION SPACE

In order to enable compilers to keep ABI compatibility for the complete x86_64 architecture, the solution needs to …

1. …make ABI breakage of derived types impossible (or obvious to the user). (cf. Section 8.3.2)

2. …keep one function call as one function call. (cf. Section 8.3.3)

3. …not require a specific $\mathcal{W}_\mathtt{T}$ from dynamically loadable libraries. (cf. Section 8.3.4)

8.4.1 DROP THE DEFAULT VECTOR TYPE

After extensive consideration and some prototyping I have not found an idea to transparently solve the issue while keeping a default vector type with varying $\mathcal{W}_\mathtt{T}$ for different microarchitectures. At this point the only solution I can conceive is a vector type that does not have a default $\mathcal{W}_\mathtt{T}$, or at least not one that follows the microarchitecture.

The C++ committee feedback on the `Vector<T>` type as specified in Chapter 4 suggested to use a policy type to select the Vector implementation instead of namespaces (cf. Chapter 6). Therefore, the following discussion refers to the following class template (with possible tag types and a portable default for x86_64):

```
namespace Vc {
  namespace Target {
    struct Scalar {};  // always present
    struct Sse2 {};    // x86(_64) specific
    struct Avx {};     // x86(_64) specific
    struct Avx2 {};    // x86(_64) specific
    typedef target_dependent Widest;  // always present
  }
  template <typename T, typename Impl = Target::Sse2> class Vector;
}
```

The user who wants to have a different default behavior can do something along the lines of:

```
template <typename T> using Vector = Vc::Vector<T, Vc::Target::Avx>;
```

Or, to get the behavior described in Chapter 4:

```
template <typename T> using Vector = Vc::Vector<T, Vc::Target::Widest>;
```

With this declaration of the default, the ABI can be stable for the complete x86_64 architecture until the user selects a specific microarchitectural subset (such as `Target::Avx`) explicitly. Thus, ABI incompatibilities will only occur after a (at least to a certain degree) conscious choice of the user. And at the same time it requires a minimal amount of code to get the behavior described in Chapter 4, which can

```
// a.cc
int_v f() [[multitarget]] {
  return int_v::IndexesFromZero();
}

// b.cc
int_v f() [[multitarget]];

int main() [[multitarget]] {
  cout << f() << '\n';
  return 0;
}
```

Listing 8.3: A very simple example using the `multitarget` attribute.

be very useful for controlled environments, such as homogeneous cluster systems and some in-house software.

8.4.2 IMPROVING MULTI-$\mathcal{W}_t$ SUPPORT

I believe it should be possible to declare a new function attribute that makes multi-SIMD-target deployment easier and at the same time can ease some bits of the ABI inconvenience.

Consider an attribute—let us call it `multitarget`—that instructs the compiler to create all possible microarchitectural variants for this function, for the given target architecture.[7] The compiler would then create a binary that can execute with the optimal $\mathcal{W}_T$ on different microarchitectures. Calls to functions with this attribute then need to be resolved at run time (via the loader or lazy), except if the call originates from another function with the `multitarget` attribute, in which case the target may already be known.

I will discuss a possible implementation and the ABI implications using the example code shown in Listing 8.3. The function `f` returns a vector object with the values `{0, 1, 2, 3, 4, …}` (depending on $\mathcal{W}_{\texttt{int}}$). When `a.cc` is compiled for x86_64, the compiler creates the following symbols:

- `_Z1fv@SSE2`
- `_Z1fv@AVX`
- `_Z1fv@AVX2`

For `b.cc`, the compiler creates the following symbols:

- `main@SSE2`
- `main@AVX`

7 I will only discuss different $\mathcal{W}_T$ to focus on the ABI issue here. It should be possible to extend the idea to transparently support different versions of SSE, too.

- `main@AVX2`

When the two TUs are linked, the call to `f` from `main` is resolved by using the same suffix. When the application is started, the loader has to resolve the call to `main`. It therefore has to execute a run-time test for the microarchitecture and then resolve the `main` function accordingly.

A function without the `multitarget` attribute can call a `multitarget` function, in which case the call has to be resolved at run time. But if the called `multitarget` function has vector types (or derived types) in its input or output, then the calling function must also be declared with `multitarget`. Otherwise, the code will only run on systems with $\mathcal{W}_{\mathtt{T}}$ equal to the one used in the non-`multitarget` function. On the other hand, this could be intended: a library uses `multitarget` functions to be generically usable, while an application linking to that library uses a fixed microarchitecture (and therefore no `multitarget` functions).

This feature eases the ABI issue because new code using vector types could easily be prepared to work across microarchitectures. This is especially important for libraries that are shipped in binary form. The ABI issue would resurface as soon as a new microarchitecture with different $\mathcal{W}_{\mathtt{T}}$ is released. At this point existing libraries will not contain support for the new ABI and fail to integrate with new code. The problem could be reduced for dynamic libraries that are linked at startup by applying the strategy described in Section 8.3.4. But as discussed before, plug-ins will break the scheme in any case.

8.5 FUTURE RESEARCH & DEVELOPMENT

This chapter has shown that there is still unclear direction how to define a portable vector type which enables compiler vendors to make a target architecture ABI-compatible throughout all SIMD variants. The requirements of C++ users certainly differ on this point and there will probably be no clear one-fits-all answer. This issue needs more experience and work in real-world use to come to a final conclusion.

9 LOAD/STORE ABSTRACTIONS

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth (1974)

Load and store operations are the main reason why a user of **`Vector<T>`** needs to be concerned with $\mathcal{W}_T$ and the differences for different target systems. Consider the typical vectorization approach of a loop transformation. In this case a loop over an array of scalar values is transformed into a loop with a stride of $\mathcal{W}_T$ and **`Vector<T>`** loads and stores inside the loop, which enable processing of $\mathcal{W}_T$ values in parallel. Because of these loads and stores, which the user calls, $\mathcal{W}_T$ surfaces into the loop stride. This is similar to portable code which requires using the `sizeof` operator in order to work correctly. This is, of course, the correct way to handle the architecture differences, but it is unfortunate that a user has to think about this problem. The vector types would be much nicer and much easier to use if $\mathcal{W}_T$ would never surface into user code. Then, the user is only concerned with expressing data-parallel processing and data-parallel storage, which is the abstraction level that is most useful for algorithm and data structure development.

In the following I will present a few ideas and approaches to solve the issue. Much of this topic is still open for research and development.

9.1 VECTORIZED STL ALGORITHMS

The C++ committee is currently developing an extension of the existing STL algorithms, which support execution in parallel. The preliminary specification [36] "describes requirements for implementations of an interface that computer programs written in the C++ programming language may use to invoke algorithms with parallel execution". As such, it deals with parallelization in terms of multi-threading and SIMD. The specification of the `std::par_vec` policy requires the compiler to

```cpp
std::vector<int> data = ...;
for_each(Vc::vec, data.begin(), data.end(), [](auto &x) {
  x = x * x + 3;
});
```

Listing 9.1: Example of the `for_each` algorithm with the `Vc::vec` policy.

treat code called from such algorithms special in such a way that "multiple function object invocations may be interleaved on a single thread". The library therefore does not actually vectorize the code, it only annotates code so that the compiler might do it.

9.1.1 VECTORIZED ALGORITHMS WITH VECTOR TYPES

There is an alternative approach using vector types, which works nicely with generic lambdas and is a useful abstraction to hide load and store functions of the vector types. Consider the example use of `for_each` in Listing 9.1. The implementation of `for_each` can call the lambda with any type that implements the required operators. Therefore, the `for_each` implementation can use the **`int_v`** type and a scalar type.[1] The scalar type can either be **`int`** or `Vc::Scalar::`**`int_v`**. The latter has the advantage that the lambda can rely on having the full Vc interface available. The need for using more than just the **`int_v`** type arises from the possibility that `data.size()` is not a multiple of $\mathcal{W}_{\mathtt{int}}$. In that case there will be a number of entries in `data` that cannot be loaded or stored as a full SIMD vector without risking an out-of-bounds memory access.[2] Additionally, an implementation of `for_each` may choose to do a prologue that processes initial elements as scalars if they are not aligned on the natural alignment of SIMD objects.

The parallel algorithms proposal [36] needs to use vector semantics (cf. Section 2.1) in the code that is called from a parallel algorithm using the `std::par_vec` policy. With the vector types solution presented here, the vectorization policy of the parallel algorithms would not need to restrict thread synchronization and exceptions.

1 This could even be taken further to additionally support smaller SIMD types. For example on an AVX2 machine, where the main vector type is `AVX2::`**`int_v`**, the next choice would be `SSE::`**`int_v`**, and finally `Scalar::`**`int_v`**.

2 The load/store problem can be solved with partial loads and stores, either as masked load/store instructions or a concatenation of partial loads / segmentation into partial stores. However, then partially unused/invalid values would have to be passed to the lambda in the unused parts of the vector. Such an implementation of the algorithm would therefore have to use **`SimdArray`**`<`**`T`**`, N>` for prologue and epilogue.

9.1.2 IMPLEMENTATION

Listing 9.2 shows a possible implementation of a vectorized `for_each`. The implementation could be generalized further to support containers that do not store their values in contiguous memory. In the same manner support for the restrictive **`InputIterator`** class of iterators can be implemented. Obviously, memory access would not use efficient vector loads and stores anymore. Note that vectorization of composite types becomes possible with the work described in Chapter 10.

9.2 VECTORIZING CONTAINER

It is possible to implement a container class template which stores the data in memory, such that it is optimized for data-parallel iterations. For fundamental arithmetic types this is as simple as ensuring alignment of the first entry and padding at the end of the container for full vector loads and stores. For structured types the `simdize<T>` helper (cf. Chapter 10) can be used to create an interleaving of the data that is appropriate for vector iterations.

A very important feature of such a container is the ability to additionally access and iterate the stored data via scalar types. This is a non-trivial problem since it requires accessing the same memory via different types, which is undefined behavior in C and C++. The container class can solve it in the same way that **`Vector<T>`** solves it for the subscript operator (Section 4.7).

Vc implements such a container for fundamental arithmetic types with the `Vc::`**`Memory`** class template. It allows both runtime sized containers as well as using a constant expression for the size. The latter enables a few optimizations since the compiler knows the loop bounds and can calculate some offset calculations at compile-time.

9.3 CONCLUSION

This chapter has discussed possible abstractions that hide the loop strides and vector loads and stores. That way the $\mathcal{W}_{\mathrm{T}}$ differences, which can make writing portable code more fragile, can be concealed from the user.

This area of research and development is still at the beginning. However, the existing abstractions show that the approach discussed here can make portable use of vector types considerably easier.

Further developments in reflection capabilities in the C++ language will also enable more powerful abstractions in this area. Therefore, serious research effort in this area should take the possibilities of reflection in C++ into account and contribute to the standardization process to ensure that a future C++ standard will be expressive enough.

```cpp
template <typename It, typename UnaryFunction>
inline enable_if<
    is_arithmetic<typename It::value_type>::value &&
        is_functor_argument_immutable<
            UnaryFunction, Vector<typename It::value_type>>::value,
    UnaryFunction>
for_each(Vc::vec_policy, It first, It last, UnaryFunction f) {
  typedef Vector<typename It::value_type> V;
  typedef Scalar::Vector<typename It::value_type> V1;
  for (; reinterpret_cast<uintptr_t>(addressof(*first)) &
                 (V::MemoryAlignment - 1) &&
             first != last;
       ++first) {
    f(V1(addressof(*first), Vc::Aligned));
  }
  const auto lastV = last - (V::Size + 1);
  for (; first < lastV; first += V::Size) {
    f(V(addressof(*first), Vc::Aligned));
  }
  for (; first != last; ++first) {
    f(V1(addressof(*first), Vc::Aligned));
  }
  return move(f);
}

template <typename It, typename UnaryFunction>
inline enable_if<
    is_arithmetic<typename It::value_type>::value &&
        !is_functor_argument_immutable<
            UnaryFunction, Vector<typename It::value_type>>::value,
    UnaryFunction>
for_each(Vc::vec_policy, It first, It last, UnaryFunction f) {
  typedef Vector<typename It::value_type> V;
  typedef Scalar::Vector<typename It::value_type> V1;
  for (; reinterpret_cast<uintptr_t>(addressof(*first)) &
                 (V::MemoryAlignment - 1) &&
             first != last;
       ++first) {
    V1 tmp(addressof(*first), Vc::Aligned);
    f(tmp);
    tmp.store(addressof(*first), Vc::Aligned);
  }
  const auto lastV = last - (V::Size + 1);
  for (; first < lastV; first += V::Size) {
    V tmp(addressof(*first), Vc::Aligned);
    f(tmp);
    tmp.store(addressof(*first), Vc::Aligned);
  }
  for (; first != last; ++first) {
    V1 tmp(addressof(*first), Vc::Aligned);
    f(tmp);
    tmp.store(addressof(*first), Vc::Aligned);
  }
  return move(f);
}

template <typename It, typename UnaryFunction>
inline enable_if<!is_arithmetic<typename It::value_type>::value,
                 UnaryFunction>
for_each(Vc::vec_policy, It first, It last, UnaryFunction f) {
  return for_each(first, last, move(f));
}
```

Listing 9.2: A possible implementation of a vectorized `std::for_each`.

10

AUTOMATIC TYPE VECTORIZATION

The only way of finding the limits of the possible
is by going beyond them into the impossible.
— Arthur C. Clarke (1962)

An important pattern in data structure design for data-parallel processing is the creation of structures with `Vector<T>` members. Thus, conceptionally the developer creates a vector variant of the structure.

Generally, one of the optimization aspects of data structure design is storage order. The three main constraints to optimize for are:

1. maintainability of the code,
2. vector loads and stores,
3. and efficient memory access (e.g. cache usage).

Classically, there are two approaches: *AoS* and *SoA* (see Figure 10.1, cf. [76]). With *AoS* the program uses *arrays of structures* (where the structures are built from fundamental arithmetic types) to store its data in memory. With *SoA* (*structures of arrays*) the storage order in memory is transposed. This latter storage order is used by many developers to improve the data structures for vector loads and stores. At the same time *SoA* often decreases locality of related objects and thus decreases cache efficiency. Furthermore, the data structures are significantly changed, from containers storing many objects into objects storing many containers. This conflicts with the typical approach of object-orientation and significantly alters the logical structure of the data.

Therefore, for vectorization, the *AoVS* data structure layout is important. In this case *arrays of vectorized structures* are defined (see Figure 10.1). This introduces the concept of vectorization of structures with several important benefits:

- It constructs chunks of memory that keep related data localized in memory, thus allowing the CPU to reuse cache line fetches more efficiently and to ease the load on the prefetcher.

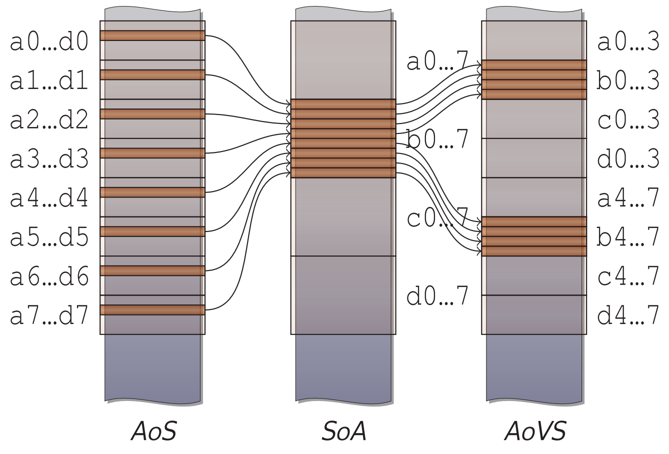


Figure 10.1: Illustration of the memory layout for different storage orders. *AoS* (array of structures) shows an array of eight objects which stores the four members `a, b, c, d`. *SoA* (structure of arrays) shows the transposition of *AoS* where a structure of four arrays is stored in memory. *AoVS* (array of vectorized structures) finally rearranges this into an array of small chunks of *SoA* storage order that are exactly as large as $\mathcal{W}_\mathtt{T}$ ($\mathcal{W}_\mathtt{T} = 4$ in this example).

- At the same time it stores the corresponding members from neighboring objects in memory in such a way that the member values of $\mathcal{W}_\mathtt{T}$ objects can be transferred efficiently to a vector register and back. This is the basis for horizontal vectorization (cf. [59]).

- *SoA* requires the use of many arrays of fundamental types. This makes aliasing related optimizations difficult for the compiler. It must assume that all pointers of the same data type may point to the same memory location. Without this restriction the compiler is able to transform the code more freely. With *AoVS* the compiler only has to assume aliasing of the outer array over the vectorized structures, which leads to more optimization opportunities.

- It is possible to access the **`Vector`**`<`**`T`**`>` objects directly in memory. With structures or arrays of scalar types a new **`Vector`**`<`**`T`**`>` object must be created, which is stored on the stack in case of register spilling or a function call. This enables the compiler to optimize memory usage register allocation.

Compared to *AoS* the maintainability still decreases, though. The logical structuring suffers in a similar way as for *SoA*, since multiple scalar objects are combined to a single vectorized object.

Maintainability of *AoVS* suffers especially because it is not sufficient for most applications to only use vectorized data structures. Therefore, many users will have to duplicate the data structures as a scalar and a vectorized variant (e.g. List-

```cpp
struct Point {
  float x, y, z;
  float length() const { return std::sqrt(x * x + y * y + z * z); }
};
struct PointV {
  float_v x, y, z;
  float_v length() const { return Vc::sqrt(x * x + y * y + z * z); }
};
```

Listing 10.1: The typical pair of data structures that appear in horizontally vectorized applications.

```cpp
template <typename T> struct Point {
  T x, y, z;
  T length() const { return std::sqrt(x * x + y * y + z * z); }
};
using Point1 = Point<float>;
using PointV = Point<float_v>;
```

Listing 10.2: With a class template the code duplication from Listing 10.1 can be avoided.

ing 10.1). The *scalar structure*, that is, a structure with member variables of fundamental arithmetic types, is duplicated as a *vectorized structure*, which provides the same members (variables and functions) but with Vc vector types replacing the scalar types. This scalar structure is necessary to support the equivalent of working with a scalar entry of a `Vector<T>` object for structured types.

Code development that follows this pattern is tedious work and error-prone because it requires significant amounts of copy-and-paste programming and all the typical programming errors resulting from only testing one of the types. Furthermore, if the developer wants to insert and extract a scalar object to/from the vectorized structure a lot of boring and repetitive work is required. This is another candidate for subtle bugs.

This chapter therefore discusses solutions that automate this data structure vectorization. The feature is limited by the capabilities of the current C++ standard. Therefore, it is important to investigate what is still missing from the language to make this feature more powerful and useful. The basic approach is the development of an expression that can automate the idea shown in Listing 10.2, where the template parameter(s) can either be a scalar or vector type. This implies a discussion of the existing reflection capabilities in C++: How can types, interfaces, and function implementations be transformed directly from C++ (i.e. without an additional "meta" compiler).

```
 1  template <typename T>
 2  struct ReplaceTypes : public std::conditional<
 3                            (std::is_same<T, short>::value ||
 4                             std::is_same<T, unsigned short>::value ||
 5                             std::is_same<T, int>::value ||
 6                             std::is_same<T, unsigned int>::value ||
 7                             std::is_same<T, float>::value ||
 8                             std::is_same<T, double>::value),
 9                            Vc::Vector<T>,
10                            T> {};
11  template <template <typename...> class C, typename... Ts>
12  struct ReplaceTypes<C<Ts...>> {
13      using type = C<typename ReplaceTypes<Ts>::type...>;
14  };
```

Listing 10.3: Basic `ReplaceTypes` class template.

10.1 TEMPLATE ARGUMENT TRANSFORMATION

Given a type `C<Types...>`, it is possible to transform this type into a new type that uses `Vector<T>` as template argument for every argument `T` that is a type supported by `Vector<T>`. To build the simplest expression we start with an alias template

```
template <typename T> using simdize = typename ReplaceTypes<T>::
                                type;.
```

The `ReplaceTypes` class template has to implement the necessary transformation. This class template can be built from two parts: First, if the template argument `T` is not a class template and the type is a valid template argument to `Vector<T>`, then the member type `type` is an alias for `Vector<T>`, otherwise it is simply `T` itself (lines 1–10 in Listing 10.3). Second, if the template argument is a class template then its template arguments must be replaced recursively by substituting each one via `ReplaceTypes` (lines 11–14 in Listing 10.3).

This simple implementation for `simdize<T>` is a solution that can automate the idea from Listing 10.2. The type `PointV` is now equal to `simdize<Point1>`. Or, to show the recursing capabilities, consider the type

```
simdize<std::tuple<std::vector<float>, std::string, std::pair<
                      double, short>>>
```

which is thus the same type as

```
std::tuple<std::vector<float_v, std::allocator<float_v>>, std::
                string, std::pair<double_v, short_v>>>.
```

The `simdize<T>` expression "greedily" replaces any occurrence of an arithmetic type (usable with Vc) somewhere in the template arguments with a SIMD type.[1]

1 Note that `std::string` is an alias for `std::basicstring<char, std::chartraits<char>, std::allocator<char>>`. Thus, if `Vector<T>` supported `char` as template argument, the `std::string` type would also be transformed, but with a questionable result. This is therefore a hint that `Vector<T>` should only support `signed char` and `unsigned char`, but not `char` itself.

This is a good start, but for `simdize<T>` to be really useful some more effort is required:

1. Ensure that the number of values in the vector types is equal for all replaced types. In the `tuple` example above, the combination of `float_v`, `double_v`, and `short_v` likely makes the whole simdized type useless. With SSE we would get $\mathcal{W}_{\texttt{float}} = 4$, $\mathcal{W}_{\texttt{double}} = 2$, $\mathcal{W}_{\texttt{short}} = 8$ and thus the type can only be used for horizontal vectorization with a SIMD width of two entries. We should therefore make use of the `SimdArray<T, N>` class template (cf. Chapter 7) to reconcile the vector widths.

2. Replace the `bool` type with a `Mask<T>` type. The template argument `T` needs to be chosen heuristically.

3. The number of entries in the mask types need to match the number of entries in the vector types as well. We need to consider the use of the `SimdMaskArray<T, N>` type for masks.

4. There needs to be an easy to use interface for converting objects of the scalar structure type `T` to objects of the vectorized structure type `simdize<T>`. This includes a simple broadcast, but also insertion and extraction of on scalar object at a given SIMD offset.

5. The C++ standard does not require an implementation to allocate correctly aligned memory with the `new` operator for over-aligned types. Thus, allocation of a `simdize<T>` (as defined above) object on the heap would likely lead to crashes. The type should therefore overload the `new` and `delete` operators to return correctly aligned memory addresses.

6. The memory alignment issue also needs to be fixed for `std::allocator<T>`. If a `simdize<T>` type is used with a type that uses `std::allocator` for memory allocation, then over-alignment might also not be supported correctly, leading to crashes. Thus, `Vc::Allocator<simdize<T>>` needs to be used instead.

7. Template parameter packs only match types. Therefore, `simdize<std::array<int, 3>>` would not be transformed. The `simdize` expression should transform class templates with non-type template parameters as well.

8. Conditional assignment of one `simdize<T>` object to another one should be possible. This should be shorthand for conditional assignment of each member variable of the objects.

9. The resulting type should provide a constant expression that identifies the SIMD vector size in the same manner that `Vc::Vector<T>::size()` does.

10.2 ADAPTER TYPE

In the following I will use these types as abbreviations for the different stages of the `simdize<T>` transformation:

`D` The scalar data structure to be transformed (e.g. `std::tuple<float, std::pair<unsigned int, int>>`).

`C` The class template underlying the type `D` (e.g. `std::tuple` and `std::pair`).

`S` The vectorized structure, which is `D` with all template arguments replaced by SIMD types (e.g. `std::tuple<float_v, std::pair<uint_v, int_v>>`).

`A` The adapter class inheriting from `S`.

The `ReplaceTypes` implementation above created a type `S` which is equal to an instance of the class template `C` with `Vector<T>` arguments. This is a limitation as soon as we want to declare additional member or non-member functions or even static member variables for this type. For member functions and variables this is obvious, since—apart from a few non-member operator alternatives—member functions must be declared within the class declaration. For non-member functions it is possible to live without a new type. However, the type `S` is a member of the namespace in which the `C` class template is declared. Argument-dependent name lookup will thus search in a namespace which is unknown to the `simdize<T>` implementation. Any non-member function thus would have to be placed in the global namespace. A derived type `A`, can be placed in a custom namespace of the `simdize<T>` implementation. Then, the non-member functions can be declared in the same namespace and thus the global namespace stays clean.

On the other hand, the new type `A` should not disable argument-dependent name lookup for functions of the original type `D`. Consider the example in Listing 10.4. The `distance` function, from line 6 is in the `Utils` namespace, but on line 15 it is called without namespace qualifier. This works as long as the template parameter `P` is a class which is either directly contained in the `Utils` namespace or one of its direct or indirect base classes is contained in the `Utils` namespace. Thus, the code in line 15 continues to compile if we modify `simdize<T>` to generate a type that is derived from `Utils::Point<T>`. We can conclude that we can create an adapter class on top of the simdized type `S`, as long as `S` is a class type (including `struct` and `union`) and contained in the adapter class via inheritance (i.e. not via composition).

The basic pattern for implementing the adapter class is shown in Listing 10.5. In this simple form the `Adapter` class template inherits from its template parameter. The logic that vectorizes the data structure is still completely implemented in `ReplaceTypes`.

```
namespace Utils {
  template <typename T> class Point {
    T x, y, z;
  };
  template <typename T>
  T distance(const Point<T> &a, const Point<T> &b) {
    return std::sqrt(a * a + b * b + c * c);
  }
}

using Point = Utils::Point<float>;

template <typename P>
float f(const P &p1, const P &p2) {
  return min_reduction(distance(p1, p2));
}
```

Listing 10.4: Non-member function relying on argument-dependent name lookup.

```
template <typename Base> class Adapter : public Base {};

template <template <typename...> class C, typename... Ts>
struct ReplaceTypes<C<Ts...>> {
  using type = Adapter<C<typename ReplaceTypes<Ts>::type...>>;
};

template <typename T> using simdize = typename ReplaceTypes<T>::type;
```

Listing 10.5: The required code for adding an adapter class on top of the simdized type.

10.2.1 VECTOR WIDTH

Once we have an adapter class **A** deriving from **S** we can easily implement the constant expression that denotes the SIMD width (cf. item 9 above). This requires a `static constexpr` member function in the **Adapter** class template. The value of this member requires a solution for the reconciliation of the vector widths (cf. item 1 above).

To determine the common vector width, a heuristic needs to be applied since there is no obvious intended width if different arithmetic types are used in the template arguments of **D**. A simple and effective strategy is the use of the natural vector width of the first **Vector**<**T**> that is substituted. This strategy allows an implementation that does a single pass over the template arguments.

The definition of such a **simdize**<**T**> implementation is shown in Listings 10.6 and 10.7. The code in Listing 10.6 extends the previous **simdize**<**T**> implementation with a new template parameter `N`:

- The new parameter `N` to **simdize** and **ReplaceTypes** selects whether **Vector**<**T**> or **SimdArray**<**T**, `N`> is used (cf. line 22).

- The **Adapter** class uses the template parameter `N` to implement the `size()` function.

```cpp
template <typename T, std::size_t N,
          bool = (std::is_same<T, short>::value ||
                  std::is_same<T, unsigned short>::value ||
                  std::is_same<T, int>::value ||
                  std::is_same<T, unsigned int>::value ||
                  std::is_same<T, float>::value ||
                  std::is_same<T, double>::value)>
struct ReplaceTypes;

template <typename T, std::size_t N = 0>
using simdize = typename ReplaceTypes<T, N>::type;

// specialization for non-simdizable types
template <typename T, std::size_t N>
struct ReplaceTypes<T, N, false> {
  typedef T type;
};

// specialization for simdizable arithmetic types
template <typename T, std::size_t N>
struct ReplaceTypes<T, N, true>
    : public std::conditional<(N == 0 || Vector<T>::size() == N),
                              Vector<T>, SimdArray<T, N>> {};

template <typename Base,  // A class template instance with vectorized
                          // template arguments
          typename Scalar,  // The same as Base except with
                            // non-vectorized template arguments
          std::size_t N>
class Adapter : public Base {
public:
  typedef Scalar ScalarType;
  static constexpr std::size_t size() { return N; }
};
```

Listing 10.6: Extending **`ReplaceTypes`** to use an equal SIMD width for all replaced types.

10.2.2 RECURSIVE TYPE SUBSTITUTION

The code in Listing 10.7 is used to recurse over the template arguments of **`D`**. The partial specialization of **`ReplaceTypes`** on line 76 uses the template class **`SubstituteOneByOne`** to substitute the template argument types. The result of this substitution is returned as a structure type with one static member for the determined vector width and a template alias that can instantiate a given class template with the substituted parameter pack. If the type substitution yields the same types (line 80) then **`ReplaceTypes`** defines its member type **`type`** to the original template argument that it was instantiated with (i.e. it does not use the **`Adapter`** class). Otherwise, the substituted type contains a vector type and therefore needs to create an instance of the **`Adapter`** class template which derives from it (line 82).

The **`SubstituteOneByOne`** class template needs two parameter packs to iterate over the template arguments. One parameter pack stores the types that are already

```cpp
// Typelist for multiple parameter packs in one class template
template <typename... Ts> struct Typelist;

// Try substituting one type after another - the first one that
// succeeds sets N (if it was 0)
template <std::size_t N, typename Replaced, typename... Remaining>
struct SubstituteOneByOne;
template <std::size_t N, typename... Replaced, typename T,
          typename... Remaining>
struct SubstituteOneByOne<N, Typelist<Replaced...>, T, Remaining...> {
private:
  // U::size() or 0
  template <typename U, std::size_t M = U::size()>
  static std::integral_constant<std::size_t, M> size_or_0(int);
  template <typename U>
  static std::integral_constant<std::size_t, 0> size_or_0(...);
  typedef simdize<T, N> V;
  static constexpr auto NN =
      N != 0 ? N : decltype(size_or_0<V>(int()))::value;

public:
  using type = typename SubstituteOneByOne<
      NN, Typelist<Replaced..., V>, Remaining...>::type;
};

// specialization for ending the recursion and setting the return type
template <std::size_t N_, typename... Replaced>
struct SubstituteOneByOne<N_, Typelist<Replaced...>> {
  // Return type for returning the vector width and list of
  // substituted types
  struct type {
    static constexpr auto N = N_;
    template <template <typename...> class C>
    using Substituted = C<Replaced...>;
  };
};

// specialization for class templates where all template arguments
// need to be substituted
template <template <typename...> class C, typename... Ts,
          std::size_t N>
struct ReplaceTypes<C<Ts...>, N, false> {
  typedef typename SubstituteOneByOne<N, Typelist<>, Ts...>::type tmp;
  typedef typename tmp::template Substituted<C> Substituted;
  static constexpr auto NN = tmp::N;
  typedef typename std::conditional<
      std::is_same<C<Ts...>, Substituted>::value, C<Ts...>,
      Adapter<Substituted, C<Ts...>, NN>>::type type;
};
```

Listing 10.7: Extending `ReplaceTypes` to use an equal SIMD width for all replaced types.

substituted and the other pack stores the types that still need to be processed.[2] The type `T` between the two lists signifies the type that is being processed by the current `SubstituteOneByOne` instance. This type is transformed via a `simdize<T, N>` expression (line 51) and thus recurses if the type `T` is a template instance. In case the `N` parameter to `SubstituteOneByOne` was zero, a successful type replacement then has to define the vector width for all remaining type replacements. Line 53 therefore checks for `N == 0`, in which case it tries to call `V::size()` as a constant expression. If this were ill-formed then a zero is returned through the SFINAE mechanism in the `size_or_0` member function overloads (lines 47–50). With `T` processed, the substitution process recurses via defining the next `SubstituteOneByOne` instance (line 56) as the member type `type`. Line 62 finally ends the recursion of `SubstituteOneByOne` when the `Remaining` parameter pack is empty. This `SubstituteOneByOne` instance then defines the return type `type` as a struct with the information that needs to be passed back to `ReplaceTypes`, which captures this type on line 77.

10.2.3 BROADCAST CONSTRUCTOR

With the adapter class `A` deriving from `S`, it is possible to add a constructor to the generated type, which allows the conversion from an object of type `D` to `S`. This is analogous to the broadcast constructor of `Vector<T>`. It converts a scalar object to an object of the vectorized type by broadcasting the scalar members into all entries of the vector members. The constructor definition is shown in line 12 of Listing 10.8. Note that the interface ensures the existence of the `std::tuple` interface for the constructor parameter type `U`. This is necessary to access the data members of the constructor argument, as shown in line 19.

10.2.4 INHERITED CONSTRUCTORS

The constructors of `C` (which are still valid and useful for the template instantiation `S`) can be inherited by `A`, so that no functionality is lost (line 2 of Listing 10.8).[3] For completeness `A` needs a defaulted default constructor (line 5), which will be usable only if `S` has a default constructor.

2 C++ cannot distinguish where one pack ends and the next starts. Therefore, the first pack is passed via the `Typelist` class template.

3 Constructor inheritance is not the right solution for aggregates. In this case no constructor would be inherited even though the base type can be instantiated via brace initialization. Alternatively, a generic constructor that forwards all arguments can be used. However, such a forwarding constructor requires complicated logic to determine whether to use parenthesis, braces, or double braces (such as for `std::array`) for the call to the base constructor.

```
  // inherit constructors from base class
  using S::S;

  // default constructor, ill-formed if S::S() is ill-formed
  Adapter() = default;

  // broadcast constructor
  template <
      typename U, typename Seq = make_index_sequence<std::tuple_size<
                      typename std::decay<U>::type>::value>,
      typename = std::enable_if_t<has_tuple_interface<U>::value>>
  Adapter(U &&x)
      : Adapter(static_cast<const D &>(x), Seq()) {}

private:
  // implements the broadcast
  template <std::size_t... Indexes>
  Adapter(const D &x, Vc::index_sequence<Indexes...>)
      : S{get<Indexes>(x)...} {}
```

Listing 10.8: The constructor definitions of the **simdize**<T> adapter class.

```
template <typename S, typename D, std::size_t N>
D simdize_extract(const Adapter<S, D, N> &a, std::size_t i);

template <typename S, typename D, std::size_t N>
void simdize_insert(Adapter<S, D, N> &a, std::size_t i, const D &x);
```

Listing 10.9: The definition of the scalar insertion and extraction functions.

10.3 SCALAR INSERTION AND EXTRACTION

The **Adapter** class implements conversion from a scalar object to a vector object. Insertion and extraction of scalar objects into/from a vector object with a given offset are still missing, though (cf. item 4 on page 117). This can be implemented via a subscript operator in the **Adapter** class. However, if **S** implements a subscript operator they would likely conflict. In general, this is true for any member function in the **Adapter** class, which is why the safest solution are a non-member functions for insertion and extraction.

Listing 10.9 shows a simple and sufficient interface for the insertion and extraction functions. The `simdize_extract` function constructs a new scalar object of type **D** from the values at vector offset `i` from the vectorized object `a`. An implementation of this function needs to read all members of `a` in order to copy this data and uses the `std::get` template function to do so. This function must be defined for the type **S**, which was vectorized with the **simdize**<T> expression. In addition, the std::**tuple_size**type must be specialized for **S**, since the function needs to know how many members it needs to read. Likewise, **D** must implement `std::get` and std::**tuple_size** for writing the members to the returned object.

The `simdize_insert` function works analogously and uses `std::get` and `std::tuple_size` to copy the member values of `x` to the members of `a` at vector offset `i`.

10.4 COMPLETE SPECIFICATION

In the same manner as shown above, the `ReplaceTypes` class template can be extended to also transform `bool` arguments to mask types. The complete specification of `simdize<T>` is as follows:

```
namespace detail {
  template <typename S, typename D, std::size_t N>
  class Adapter : public S {
  public:
    typedef D ScalarType;
    static constexpr std::size_t size() { return N; }
    using S::S;
    Adapter() = default;
    template <typename U, typename = std::enable_if_t<
                              has_tuple_interface<U>::value>>
    Adapter(U &&x);
  };
  template <typename S, typename D, std::size_t N>
  D simdize_extract(const Adapter<S, D, N> &a, std::size_t i);
  template <typename S, typename D, std::size_t N>
  void simdize_insert(Adapter<S, D, N> &a, std::size_t i, const D &x);
}

template <typename T, std::size_t N = 0, typename MT = void>
using simdize = /*see below*/;
```

The alias template `simdize<T, N, MT>` denotes a type that is one of the following:

`Adapter<C<simdize<Ts, N, MT>...>, N>`

…if `T` is a class template `C<Ts...>` without non-type template parameters. If `N` is zero the first `simdize<U, 0, MT>` expression that yields a type with a static `constexpr` member function named `size()` defines `N` and `MT` for all subsequent `simdize` expressions in the pack expansion:

- `N` is set to `simdize<U, 0, MT>::size()`.
- If `U` is `bool` and `MT` is not `void` then `MT` stays unchanged.
- If `U` is `bool` and `MT` is `void` then `MT` is set to `float`.
- Otherwise `MT` is set to `U`.

`Adapter<C<simdize<U, N, MT>, Values...>, N>`

…if `T` is a class template `C<U, Values...>` with non-type template parameters of a single integral type (`Values`). If `N` is zero the `simdize<U, 0, MT>`

expression sets the template parameter `N` of **`Adapter`** to **`simdize`**`<`**`U`**`, 0, `**`MT`**`>` `::size()`.

`Vc::Vector<T>`
: …if **`T`** is one of the arithmetic types supported by `Vc::`**`Vector`** and `N` is either zero or equal to `Vc::`**`Vector`**`<`**`T`**`>::size()`.

`Vc::SimdArray<T, N>`
: …if **`T`** is one of the arithmetic types supported by `Vc::`**`Vector`** and `N` is neither zero nor equal to `Vc::`**`Vector`**`<`**`T`**`>::size()`.

`Vc::Mask<MT>`
: …if **`T`** is **`bool`**, **`MT`** is not **`void`**, and `N` is either zero or equal to `Vc::`**`Mask`**`<`**`MT`**`>` `::size()`.

`Vc::Mask<float>`
: …if **`T`** is **`bool`**, **`MT`** is **`void`**, and `N` is either zero or equal to `Vc::`**`Mask`**`<`**`float`**`>` `::size()`.

`Vc::SimdMaskArray<MT, N>`
: …if **`T`** is **`bool`**, **`MT`** is not **`void`**, and `N` is neither zero nor equal to `Vc::`**`Mask`**`<`**`MT`** `>::size()`.

`Vc::SimdMaskArray<float, N>`
: …if **`T`** is **`bool`**, **`MT`** is **`void`**, and `N` is neither zero nor equal to `Vc::`**`Mask`**`<`**`MT`**`>::` `size()`.

`T`
: …otherwise.

10.5 CURRENT LIMITATIONS

To use the full capabilities of the **`simdize`**`<`**`T`**`>` expression, data structure design is required to follow a specific pattern. This limits the range of application for **`simd-ize`**`<`**`T`**`>`.

- **`simdize`**`<`**`T`**`>` depends on a specific pattern of data structure definition and associated meta-data/-functions. Since the `std::`**`tuple`** class template fully implements such a pattern the **`simdize`**`<`**`T`**`>` solution requires the `std::`**`tuple`** interface. This makes `std::`**`tuple`**, `std::`**`pair`**, and `std::`**`array`** immediately usable with **`simdize`**`<`**`T`**`>`.

- Users have to define their data structures as template classes (or structs), where all data member types are template parameters of the class. Additionally, the user has to implement `get<N>` and `std::tuple_size` in order to activate the conversion functions between scalar to vector objects. The latter effectively provides a "back-door" to accessing the `private` (and `protected`) data members.

- The `simdize<T>` solution is limited to data structures without non-type template parameters.[4] Support for every non-type template parameter combination needs a new specialization in the `simdize<T>` implementation. Therefore, only a limited set of non-type template parameter class templates will be supported. Most importantly, `std::array` is supported.

- Functions that use such a vectorizable data structure must be defined as template functions and therefore either be defined in header files (`inline` or unnamed namespace) or explicitly instantiated in the source file.

10.6 CONCLUSION

This chapter has presented a solution for automatic vectorization of data structures and the template functions that work with these types. The `simdize<T>` expression helps users to create data structures that deliver the best compromise between data locality and efficient vector loads and stores (or *AoS* vs. *SoA*). If a user can follow the required pattern for data structure design, (s)he will have a powerful abstraction mechanism available for expressing both scalar and vector operations. Chapter 11 shows an example where `simdize<T>` enables generic vectorization without full knowledge about the user's data structure.

4 This is due to a limitation in variadic templates in current C++.

Part III

Vectorization of Applications

11 NEAREST NEIGHBOR SEARCHING

Search algorithms are a fundamental class of algorithms in computer science (cf. [54]). In general, a search algorithm searches through a set of values for some value that matches a search predicate.

The most significant optimization for search algorithms is a reduction of the complexity. Typically, a linear search ($\mathcal{O}(N)$) can be improved to a binary search ($\mathcal{O}(\log N)$). This requires the set of search values to be structured in a supporting manner for the search algorithm. For example, a binary tree data structure is used because it allows simple lookup with $\mathcal{O}(\log N)$ complexity.

Nearest neighbor search algorithms are a specialization using a search predicate evaluating a distance function d. The entry in the set that produces the smallest value for the distance function is the nearest neighbor and thus the result of the search. With one-dimensional search keys and a sorted set, nearest neighbor search can be as simple as a binary search using the less operator to find the two neighboring elements where the return value of the less comparison differs. Of those two elements, the element with the shorter distance is the correct search result.

If the search keys are multi-dimensional, the problem is not as simple anymore. Consideration of only a single dimension cannot yield any useful conclusion about the actual distance of the entry in the set. Therefore, there is no natural sorting order that can make searching as simple as in the one-dimensional case. Nevertheless, there are data structures (e.g. k-d tree [6], quad tree [22], …) improving nearest neighbor searches with multi-dimensional keys significantly, achieving a complexity of $\mathcal{O}(\log N)$.

Complexity is not the only important characteristic of the efficiency of a given search algorithm. In some applications, the value for N is known to be bounded. Then it is important to compare the actual run time of an $\mathcal{O}(N)$ algorithm against an $\mathcal{O}(\log N)$ algorithm. Generally, the simple, brute-force approach is faster for small values of N. This is due to more efficient memory access patterns (linear instead of random) and less overhead for address calculation. For some $N = N_0$ the $\mathcal{O}(N)$ and $\mathcal{O}(\log N)$ algorithms will be equally fast, and consequently the $\mathcal{O}(\log N)$ algorithm will be more efficient for $N > N_0$.

```
1  template <class Iterator, class T>
2  Iterator find(Iterator first, Iterator last, const T &value) {
3    for (; first != last; ++first) {
4      if (*first == value) {
5        break;
6      }
7    }
8    return first;
9  }
```

Listing 11.1: A typical `std::find` implementation.

```
1  template <class Iterator, class T>
2  Iterator find(Iterator first, Iterator last, const T &value) {
3    typename simdize<Iterator>::value_type value_v = value;
4    for (simdize<Iterator> it = first; it < last; ++it) {
5      const auto mask = *it == value_v;
6      if (any_of(mask)) {
7        return it.scalar() + mask.indexOfFirst();
8      }
9    }
10   return last;
11 }
```

Listing 11.2: `Vc::find` implementation using a **`simdize<T>`** implementation extended for random-access iterator [48, §24.2.7] types.

11.1 VECTORIZING SEARCHING

To show the applicability of vectorization to searching in general, I reimplemented the `std::find` algorithm (cf. [48, §25.2.5] and Listing 11.1) with Vc. This implementation is presented in Listing 11.2. The `find` algorithm iterates over a given iterator range (`first` to `last`) and compares each value against the search value. If the value matches, the iterator is returned. If no match is found, the `last` iterator is returned. This algorithm is a classical linear search with $\mathcal{O}(N)$ complexity.

11.1.1 LOOP-VECTORIZATION FAILS

The search loop is not loop-vectorizable (i. e. not auto-vectorizable) because loop-vectorization requires a *countable* loop (cf. Section 2.1). With the `break` statement in line 5 the iteration count is input-data dependent. In order to vectorize the search loop, the algorithm has to compare $\mathcal{W}_{\mathtt{T}}$ values per iteration; and thus potentially execute $\mathcal{W}_{\mathtt{T}} - 1$ comparisons more than necessary. To a developer it is obvious that this is efficient and correct, but to the compiler this is an incorrect translation of the source code and it would have to prove that it is allowed to do it under the "as-if" rule of C++ [48, §1.9 p1].

11.1.2 EXPLICIT VECTORIZATION

The vectorization of the `std::find` algorithm with Vc (Listing 11.2) is a straightforward transformation from iterating over one entry at a time to processing $\mathcal{W}_{\mathtt{T}}$ entries per iteration. For this example I have extended **`simdize<T>`** (cf. Chapter 10) to recognize iterator types and in particular random-access iterators (**`random_access_iterator_tag`**, cf. [48, §24.4.3]). The iterator type is then adapted to return **`simdize<iterator::value_type>`** from `operator*` and move the scalar iterator by $\mathcal{W}_{\mathtt{T}}$ with the increment and decrement operators. The comparison operators of the adapted iterator ensure that both iterators use the same alignment and thus no out-of-bounds access will be allowed.

The `Vc::find` implementation is slightly simplified since it assumes that the distance of `first` to `last` is a multiple of $\mathcal{W}_{\mathtt{T}}$. To account for this possibility the implementation would need an epilogue, and possibly a prologue, which processes the remainder values correctly (e.g. as scalars).

In the loop body the `find` algorithm has to execute the equality comparison, which produces a mask object (line 5). If any of the values in the comparison are equal, then the mask has at least one `true` value and thus the `any_of` reduction returns `true`. At this point the algorithm needs to return the (scalar) iterator of this value. The vectorized iterator returns the iterator value to the first scalar with the `scalar()` member function. The index of the first `true` value in the mask determines the additional offset, which is used to advance the scalar iterator accordingly (line 7). If no value in the complete set matches, the algorithm returns `last`, as in the `std::find` implementation.

11.1.3 BENCHMARK

The following benchmark compares the efficiencies of the `Vc::find` and `std::find` implementations. The benchmark was compiled with GCC version 4.8.2 using its libstdc++ implementation of `std::find` (which is an optimized implementation compared to Listing 11.1). The optimization relevant compiler flags used were:

```
-O3 -ffp-contract=fast -march=core-avx-i -DNDEBUG.
```

The benchmark code (cf. Appendix B) executes 10,000 calls to `find`. The search values are picked from random locations in the haystack using a uniform distribution. The 10,000 searches are repeated at least 100 times (more often for smaller haystacks) and the execution times of each repetition are recorded. Afterwards these values are used to calculate the mean and standard deviation[1] of the `find`

1 This assumes that execution time follows a normal distribution, which is not exactly true. However, it works well enough for approximating the error.

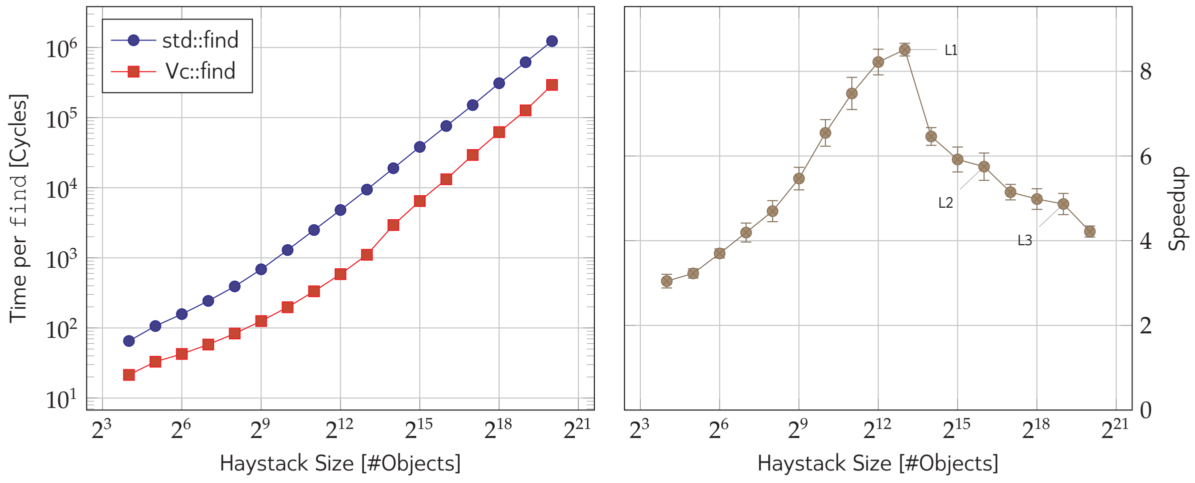


Figure 11.1: Benchmark results of a reimplementation of `std::find` with Vc (Listing 11.2). The benchmark (cf. Appendix B) uses differently sized haystacks containing random **`float`** values (x axis). The average number of CPU cycles required to execute `std::find` or a vectorized `find` are plotted in the first diagram. The second plot shows the quotient (speedup factor) of the two graphs on the left. The benchmark was executed on an Intel i5 3360M CPU with 2.80 GHz ("Turbo Mode" and power management were disabled—cf. Appendix D—) using the AVX implementation of Vc.

function's run time. If the relative error is larger than 5% the measurement is repeated, because the system was likely active with different tasks that competed for execution resources.

11.1.4 RESULTS

Figure 11.1 shows the result of the benchmark. It is clearly visible that vectorization improves the efficiency of the `find` implementation by at least a factor 3 up to slightly more than $\mathcal{W}_{\texttt{float}} = 8$. The largest speedup is at $2^{13} \cdot 4$ Bytes = 32 KiB[2]. The CPU that executed the benchmark has 32 KiB of L1 cache, 256 KiB = $2^{16} \cdot 4$ Bytes of L2 cache, and 3 MiB = $2^{19} \cdot 1.5 \cdot 4$ Bytes of L3 cache. The three cache sizes can be found in the "Speedup" plot of Figure 11.1, limiting the vectorization improvement for large data sets.

2 The values are of type **`float`** and thus 4 Bytes large.

```
template <class Iterator, class V>
std::array<Iterator, V::size()> find_parallel(Iterator first,
                                              Iterator last,
                                              const V &value) {
  std::array<Iterator, V::size()> matches;
  for (auto &x : matches) {
    x = last;  // initialize with "not "found
  }
  typename V::mask_type found(false);
  for (; first < last; ++first) {  // scalar iteration over haystack
    const auto mask = *first == value && !found;
    if (any_of(mask)) {
      found |= mask;  // keep track of the lanes that are done
      for (int i : where(mask)) {
        matches[i] = first;  // the iterator for the return value
      }
      if (all_of(found)) {
        break;  // all lanes found a match, return
      }
    }
  }
  return matches;
}
```

Listing 11.3: Alternative vectorization of the `find` algorithm using a different vectorization direction.

11.1.5 VECTORIZATION DIRECTION

There is an alternative vectorization for the `find` algorithm that executes the search loop for multiple lookups in parallel (cf. Listing 11.3). In this case the `value` parameter is a vector type and the iteration over the haystack is done with scalar iterators. This approach also speeds up the `find` algorithm compared to the `std::find` implementation, but not as efficient as the vectorization shown above. This is due to the implementation having to keep track of the vector lanes that already had a hit, in order to return the iterator to the first match. In addition, the vector usage is suboptimal since one lane after another becomes unused as the search values are found. This is different for the implementation above, where only the last vector compare in the iteration potentially executes unused compare operations.

11.1.6 THE MERIT OF VECTORIZATION

Vectorization of search algorithms cannot improve the complexity of the algorithms. However, it can make the actual implementation perform more efficiently. SIMD instructions improve load throughput, reduce branching, improve compare operations throughput, and can potentially reduce the number of pointer chases necessary to arrive at the answer. The effect is most visible for algorithms which are not slowed down by memory latency (by avoiding pointer chasing[3] or hiding the latency efficiently).

11.2 VECTORIZING NEAREST NEIGHBOR SEARCH

Having shown that vectorization of search algorithms can generally improve the efficiency of the implementation, let us investigate nearest neighbor search algorithms. As for the vectorization above, we have two principal choices of vectorization:

1. Execute the search for several search keys in parallel.
2. For one search key compare several entries of the haystack in parallel.

It may additionally be possible to vectorize the compare operation (vertical vectorization [59]) since the evaluation of the distance function consists of multiple scalar operations. This document will not cover vertical vectorization of this problem any further, because it does not scale and competes with ILP[4] (cf. Section 1.6.1) and therefore is not a useful transformation.

11.2.1 VECTORIZATION APPROACH

The two remaining choices were shown to have a clear winner above (cf. Section 11.1.5). There is no apparent reason why approach 1 would work better for an exhaustive nearest neighbor search. For a search in a search tree, the situation appears to be fundamentally different. These data structures are built in such a way that each scalar entry on a tree node needs to be fully evaluated to determine the next node. Thus, vectorization seems to require approach 1. For the k-d tree algorithm, approach 1 has been researched for raytracing on GPUs before (cf. [37, 27]).

Searching is very sensitive to load performance since the whole task of the search algorithm is to walk through a data structure to identify the match. Thus, a good algorithm does as much as possible per load, per cache line, and per pointer chase. Likewise, a good implementation of a given algorithm tries to optimize the same issues. Vectorization approach 1 for a non-exhaustive search can improve memory load efficiency only in special cases (i.e. if the matches are closely localized). However, in most cases the matches to the search keys will be stored in unrelated locations of the data structure. Then the loads cannot be coalesced, resulting in a load performance similar to the scalar implementation.

3 The term “pointer chasing” is used to describe dereferencing a pointer to load a pointer which, in turn, is dereferenced to load the next pointer, and so on … This is a typical implementation of container iteration for linked data structures such as linked lists and trees.

4 Instruction Level Parallelism

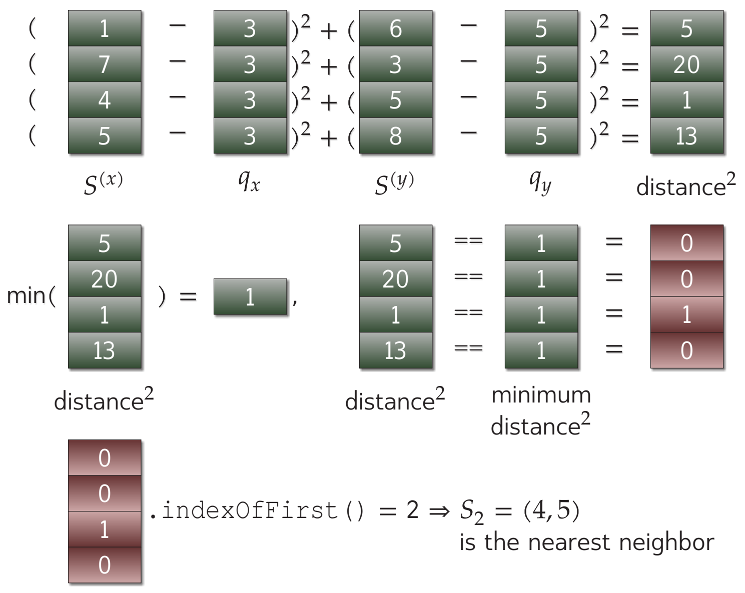


Figure 11.2: SIMD algorithm for nearest neighbor search for $q = (3,5)$ in a set $S = \{(1,6),(7,3),(4,5),(5,8)\}$ for a system with $\mathcal{W}_{\mathtt{T}} = 4$.

Consequently, the rest of the chapter focuses on approach 2. The search algorithm therefore must be able to execute comparisons for multiple values in the haystack for a single input in parallel. Section 11.3.5.1 quantifies the differences of approaches 1 and 2.

11.2.2 LINEAR NEAREST NEIGHBOR SEARCH

Consider the simple case of four two-dimensional values in the haystack and a system with $\mathcal{W}_{\mathtt{float}} = 4$. Then the algorithm simply calculates the distance for all four values in parallel. It then determines the vector entry with the minimum value, which identifies the nearest neighbor (Figure 11.2).

This idea can be extended for implementing a complete vectorized linear nearest neighbor search algorithm as shown in Listing 11.4. To showcase how simple the Vc SIMD implementation is, Listing 11.5 shows a scalar implementation of the linear nearest neighbor search algorithm.

The algorithm expects at least one entry in the `vector` argument for simplifying the return semantics. This precondition is verified first by the function. If the condition is not met it throws an exception. The first element of the input data initializes the nearest neighbor candidate variable. Subsequently, all remaining elements in the input data are compared against the candidate. If a point with a smaller distance is found, the candidate is updated.

```cpp
template <typename T>
T findNearest(const T &x_,
              const std::vector<simdize<T>> &data) const {
  auto it = data.begin();
  const auto end = data.end();
  if (it == end) {
    throw std::invalid_argument("data must not be empty");
  }
  const simdize<T> x(x_);
  simdize<T> candidate = *it;
  auto bestDistance = distance(x, *it);
  for (++it; it != end; ++it) {
    const auto tmp = distance(x, *it);
    where(tmp < bestDistance) | candidate = *it;
    where(tmp < bestDistance) | bestDistance = tmp;
  }
  return simdize_extract(
      candidate, (bestDistance.min() == bestDistance).indexOfFirst());
}
```

Listing 11.4: Vectorized implementation of the linear search algorithm in Listing 11.5.

```cpp
template <typename T>
T findNearest(const T &x, const std::vector<T> &data) {
  auto it = data.begin();
  const auto end = data.end();
  if (it == end) {
    throw std::invalid_argument("data must not be empty");
  }
  T candidate = *it;
  auto bestDistance = distance(x, *it);
  for (++it; it != end; ++it) {
    const auto tmp = distance(x, *it);
    if (tmp < bestDistance) {
      candidate = *it;
      bestDistance = tmp;
    }
  }
  return candidate;
}
```

Listing 11.5: Simple linear nearest neighbor search implementation.

```
template <typename T>
T distance(const Point<T, 3> &p0, const Point<T, 3> &p1) {
  const auto dx = p0[0] - p1[0];
  const auto dy = p0[1] - p1[1];
  const auto dz = p0[2] - p1[2];
  return dx * dx + dy * dy + dz * dz;
}
```

Listing 11.6: A possible implementation of a generic `distance` function.

The SIMD implementation requires one additional idea: The algorithm determines $\mathcal{W}_T$ candidates from disjunct and interleaved subsets of the input data. At the end of the loop the resulting best candidates are compared and reduced to a single one (in the same manner as Figure 11.2), which is the return value. Thus, horizontal reduction over SIMD vectors happens only at the very end of the algorithm.

Note that the data type for the input data is different in the scalar and SIMD implementation. The vectorized function expects a **`vector`** of vector objects of type **`simdize`**`<T>` (cf. Chapter 10), where **`T`** is the type of the search point.

11.2.3 PROGRAMMABILITY

Consider how few modifications are necessary to arrive at the vectorized `findNearest` implementation starting from Listing 11.5. Most importantly, a developer needs to understand masks and write-masking and how **`simdize`**`<T>` can be used to create vectorized generic interfaces. The `distance` function, which is called from both `findNearest` implementations, needs to be vectorizable, though. This is actually very easy with a generic implementation, such as shown in Listing 11.6.

11.2.4 BENCHMARK

The two `findNearest` functions can be benchmarked analogously to the benchmark for `std::find` (cf. Section 11.1.3). Appendix C presents the full benchmark code. The benchmark was compiled with GCC version 4.8.2. The optimization relevant compiler flags used were:

`-O3 -ffp-contract=fast -march=core-avx-i -DNDEBUG`.

The benchmark measures the run times of insertion and searching for the scalar and vectorized linear nearest neighbor search data structures (**`LinearNeighborSearch`** and **`LinearNeighborSearchV`**). These two classes implement the member function `findNearest` as shown in Listing 11.4 and Listing 11.5. The benchmark iterates over different haystack sizes and executes 10,000 nearest neighbor lookups for each. The statistics are determined in the same way as described in Section 11.1.3.

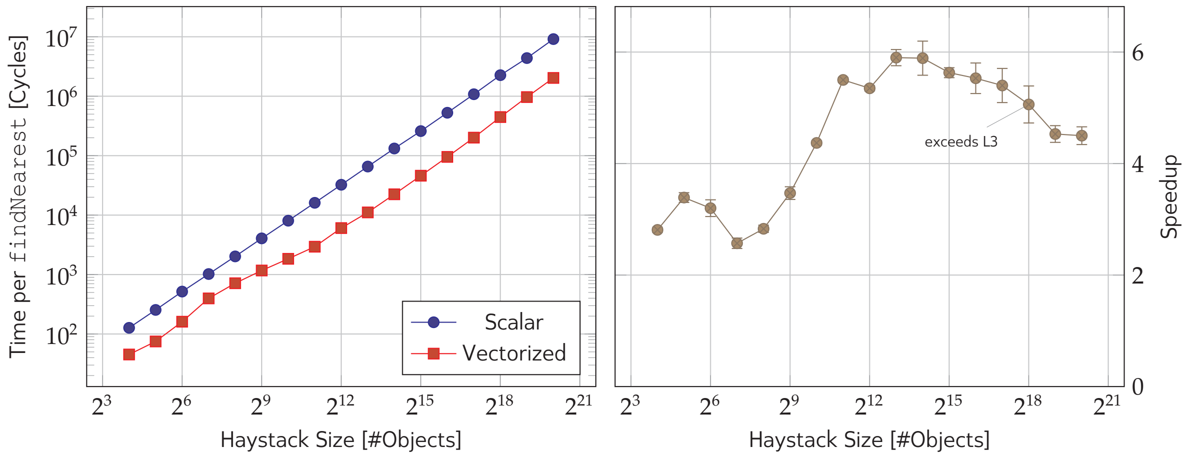


Figure 11.3: Benchmark results of linear nearest neighbor searching with the implementations Listing 11.4 and Listing 11.5.

The point objects used in the benchmark are three-dimensional points using single-precision floating point variables (12 Bytes per point object) and a distance function as shown in Listing 11.6. Most importantly, this `distance` function omits the square-root function since searching for the nearest neighbor does not depend on the actual magnitude but only on a correct total order.

11.2.5 RESULTS

The vectorized search is consistently more efficient than the scalar variant as Figure 11.3 shows. The first plot shows the average number of CPU cycles a single function call required to execute the scalar or vectorized `findNearest` functions. The values on the x-axis need to be multiplied by 12 Bytes to determine the size of the data set in Bytes. Thus, the smallest search iterated over 96 Bytes and the largest search over 12 MiB. A run time of 10^7 cycles multiplied by 10,000 searches and 100 repetitions (for the statistics) at 2.8 GHz equates to $\frac{10^{7+4+2}}{2.8 \cdot 10^9\ \text{Hz}} \approx 1\ \text{h}$. The second plot in Figure 11.3 shows the quotient of the run time of the scalar function divided by the run time of the vectorized function (speedup factor). It clearly shows that vectorization improves the efficiency of the exhaustive search considerably.

The benchmark was executed on an Intel i5 3360M CPU with 2.80 GHz (“Turbo Mode” and power management were disabled—cf. Appendix D—) using the AVX implementation of Vc. At $2^{18} \cdot 12$ Bytes $= 3$ MiB haystack size, the L3 cache is exhausted and memory performance decrease the achievable speedup. The speedup of the vectorized algorithm is still a factor of four for the largest benchmarked data set, though, showing the usefulness of vectorization for exhaustive nearest neighbor search implementations.

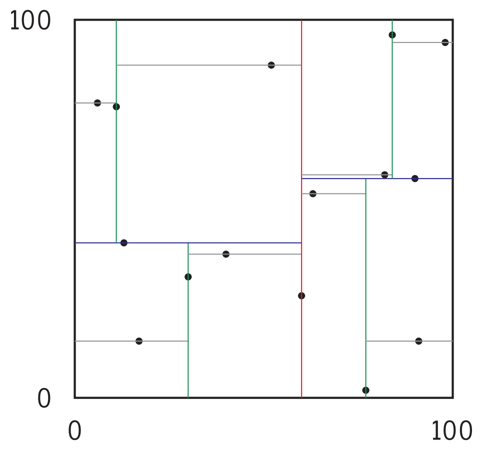


Figure 11.4: k-d tree decomposition for the point set (17, 15) (30, 32) (40, 38) (13, 41) (6, 78) (11, 77) (52, 88) (60, 27) (63, 54) (77, 2) (91, 15) (90, 58) (82, 59) (84, 96) (98, 94).

Inserting new points into the search data structure is slightly slower for the vectorized variant, though. This is obvious if you consider that inserts of scalar structures need to be adapted for a `std::vector<simdize<T>>` container. This requires the `simdize_insert` function (cf. Section 10.3) to change the storage order to *AoVS*. Benchmarking insert performance shows that the scalar data structure requires 65%–85% of the run time of the vectorized data structure. Note, though that this overhead is only necessary if the interface needs to use scalar structures. If the program already uses *AoVS* storage the insert can be even more efficient since it then uses vector loads and stores for copying the data.

11.3 K-D TREE

Finally, this section will show how the vectorization approach from the previous section can be applied to the k-d tree algorithm. Bentley [6] introduced the k-d tree data structure in 1975. The data structure is a binary search tree enabling nearest neighbor search with $\mathcal{O}(\log N)$ complexity [28].

Each node of the k-d tree stores a single point. The depth of the node in the tree determines which dimension of the multi-dimensional point is used as discriminator for traversal. Consider the example shown in Figure 11.4. There are 15 points stored in the k-d tree. After sorting for the first dimension, the (60, 27) point is at the center and therefore chosen for the root node of the tree. All points with $x < 60$ are inserted on the left child node and points with $x > 60$ are inserted on the right child node.[5] From this construction, the tree contains seven points on the left and right subtrees. The next node is discriminated via the second dimension. Therefore, the left child node stores the (13, 41) point and the right child node stores the (90, 58) point. Since k-d tree structure in this example only uses two dimensions, the next level of the tree discriminates on the first dimension again.

This data structure leads to inserts with $\mathcal{O}(\log N)$ complexity (without balancing). Most importantly, nearest neighbor lookup is now possible with $\mathcal{O}(\log N)$ complexity.

11.3.1 K-D TREE NEAREST NEIGHBOR LOOKUP

Given a search point $p = (p_x, p_y)$ nearest neighbor lookup traverses the tree down to a leaf node using the same rules as described above for insertion. This leaf node then is used as a first candidate for the solution. The algorithm then walks the tree back up to the root node. For every node it checks whether the hypersphere around the search point, with radius equal to the distance to the candidate point, intersects the splitting plane. If it does intersect, the algorithm must check on the other side of the splitting plane for a closer match.

The k-d tree nearest neighbor search algorithm is inherently not vectorizable with this original definition of the tree structure. One might consider speculative calculation of the distance for multiple nodes down the tree to create a vector of points from the data set. However, this makes the implementation complex and unlikely to increase efficiency.

11.3.2 K-D TREE VECTORIZATION

In order to efficiently vectorize, the k-d tree data structure must be modified. The idea is to create nodes that store $\mathcal{W}_{\mathrm{T}}$ points instead of a single one. One node hence stores one **`simdize<T>`** object instead of one **`T`** object (cf. Listing 11.7). (The original k-d tree structure uses one object of **`T`** per node.) This requires changes to the insert and nearest neighbor search algorithms as presented in the following.

The following algorithm descriptions use the function $K_D(x)$, which returns the discriminator of the object x for the D^{th} dimension (most commonly K_D returns the D^{th} coordinate of a point, e.g. $K_0((7,5)) = 7$ and $K_1((7,5)) = 5$). The **`Node`** structure contains four members: a vector of data (*data* of type **`simdize<T>`**), two pointers to the child nodes (*lo* and *hi*), and an integer *entries* tracking the last fill index of *data*.

5 Bentley [6] originally suggested that points with equal discriminator use the concatenation of all dimensions to determine where the point needs to be inserted. But other strategies, such as always choosing left or right or alternating left/right according to some heuristic are also conceivable and often more efficiently implementable.

```cpp
template <typename T,
          std::size_t Dimensions = std::tuple_size<T>::value>
class KdTreeV {
  using V = simdize<T>;
  template <std::size_t SplittingPlane> struct Node : public V {
    std::array<
        std::unique_ptr<Node<(SplittingPlane + 1) % Dimensions>>, 2>
        child;
    unsigned int entries;
    void insert(const T &);
    void findNearest(const V &, Candidate &) const;
  };
  std::unique_ptr<Node<0>> root;

public:
  void insert(const T &x);
  T findNearest(const T &x) const;
};
```

Listing 11.7: The structure of the vectorized k-d tree class.

11.3.2.1 THE *INSERT* ALGORITHM

INSERT$(x, Q = ROOT, D = 0)$:

```
 1: if $Q = \Lambda$ then
 2:     $Q \leftarrow$ new Node
 3:     for $i = 0$ to $\mathcal{W}_T - 1$ do
 4:         $Q.data_i \leftarrow x$
 5:     end for
 6:     $Q.hi \leftarrow \Lambda$
 7:     $Q.lo \leftarrow \Lambda$
 8:     $Q.entries \leftarrow 1$
 9:     return
10: end if
11: if $Q.entries < \mathcal{W}_T$ then
12:     $Q.data_{Q.entries} \leftarrow x$
13:     $Q.entries \leftarrow Q.entries + 1$
14:     return
15: end if
16: if $K_D(Q.data_i) < K_D(x) \forall_i$ then
17:     $Q \leftarrow Q.hi$
18: else if (not $K_D(Q.data_i) < K_D(x)) \forall_i$ then
19:     $Q \leftarrow Q.lo$
20: else
21:     if $count(K_D(Q.data_i) < K_D(x)) >= \mathcal{W}_T/2$ then {majority in $Q$ smaller than
        $x$}
22:         for all $d$ in $K_D(Q.data_i)$ do
```

```
23:         if $d$ is maximal then
24:             $R \leftarrow i$
25:         end if
26:     end for
27:     $Q.data_R \leftrightarrow x$
28:     $Q \leftarrow Q.hi$
29:   else
30:     for all $d$ in $K_D(Q.data_i)$ do
31:         if $d$ is minimal then
32:             $R \leftarrow i$
33:         end if
34:     end for
35:     $Q.data_R \leftrightarrow x$
36:     $Q \leftarrow Q.lo$
37:   end if
38: end if
39: $D \leftarrow D + 1 \mod Dimensions$
40: $INSERT(x, Q, D)$
```

On *INSERT*, the algorithm first checks whether the node Q is fully filled (cf. lines 1–15). If not, x is inserted into the node and the algorithm terminates. Otherwise, if *INSERT* is called with a full node ($Q.entries = \mathcal{W}_{\mathrm{T}}$) the algorithm must determine where x needs to be stored. There are three choices: to the *lo* child node, to the *hi* child node, or to this node. The first two choices are analogous to the original k-d tree insert algorithm, only that the condition requires all discriminators of *Q.data* to agree on *hi* or *lo*. If they do not agree then $\min_i(K_D(Q.data_i)) < x \leq \max_i(K_D(Q.data_i))$.[6] One of the entries in Q then needs to be swapped with x before the recursion continues to a child node. The algorithm determines whether the majority of discriminators agree on *hi* (line 21). If yes, the entry in *Q.data* with maximum discriminator is swapped with x and *INSERT* recurses to *Q.hi*. Otherwise, the entry in *Q.data* with minimum discriminator is swapped with x and *INSERT* recurses to *Q.lo*. Alternatively, the mean of the discriminator values in the node Q can be used to decide on the direction.

Note that on node creation not only the first entry in the vector *data* is initialized; rather the object x is broadcast to all entries of the *data* vector (cf. line 3). This ensures that all entries contain valid values for subsequent vector compares. For

6 Actually, the algorithm is defined such that only the less-than operator is used. This leads to the $x \leq \max$ relation.

INSERT this is not important because compares are only executed on fully filled nodes. But the *FINDNEAREST* algorithm depends on it.[7]

A possible optimization modifies the *INSERT* algorithm such that records in the nodes are sorted according to their discriminator value. This enables quick retrieval of the leftmost and rightmost discriminator values without the use of vector reduction operations in *FINDNEAREST*.

After insertion of the same 15 points as for Figure 11.4 the vectorized tree structure looks as in Figure 11.5. The root node (in red) contains four points and thus partitions the complete area into three subareas. A C++ implementation of the *INSERT* algorithm is presented in Appendix E.

11.3.2.2 the *findnearest* algorithm

The vectorized *FINDNEAREST* algorithm uses the same idea as originally defined for the optimized k-d tree [28].[8] The algorithm must descend the tree according to the discriminator $K_D(x)$ on each node until it reaches a leaf node. It determines the best candidate of a leaf node using the algorithm from Figure 11.2. For leaf nodes, it is important that none of the entries yield incorrect answers and therefore must use valid records. This was ensured by the broadcast of the *INSERT* algorithm on node construction. Effectively, a leaf node that is not fully populated will calculate the same distance multiple times for the exact same data. Only the first minimum distance in the vector will be used by the algorithm; therefore this works fine without extra masking support.

The choice for descending on the *hi* or *lo* child is not as obvious as with the scalar k-d tree, though, because there are multiple discriminator values stored in each node ($K_D(data_i)$). This is visible in Figure 11.5. The root node (in red) stores four records: $data_{0...3}$ = (40, 38) (52, 88) (60, 27) (63, 54). Since the root node discriminates on the x-coordinate, there is one record left of the search point (50, 5) and three records on the right. In this case, the search point's discriminator value lies between the minimum and maximum of the node's discriminator values ($\min(K_D(data)) < K_D(x) < \max(K_D(data))$) and it appears like an arbitrary choice whether to first descend the *lo* node or the *hi* node. The best choice for optimization in the tree ascent stage is to use the center between the minimum and maximum discriminator values of the current node ($\frac{\min(K_D(data))+\max(K_D(data))}{2}$). Therefore, the algorithm aliases the *hi* and *lo* children as *near* and far. The *near* child is set to *hi* if the discriminator of the search point is larger than the center discriminator of the current node; otherwise it is set to *lo*. The far child references the opposite node.

7 In fact, *FINDNEAREST* can be defined in such a way that the broadcast is unnecessary. This would imply extra overhead for the algorithm, though.

8 The optimized k-d tree stores records only in leaf nodes; non-leaf nodes are only used for partitioning. The leaf nodes are defined as *buckets* and thus store multiple records and require a linear nearest neighbor search such as discussed in Section 11.2.2.

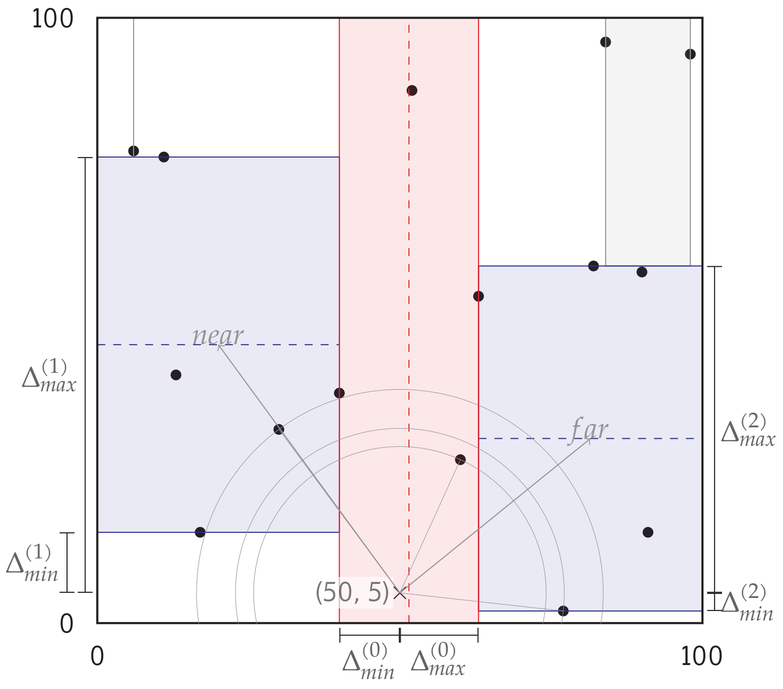


Figure 11.5: Vectorized k-d tree decomposition with $\mathcal{W}_{\mathrm{T}} = 4$ for the same data set as used in Figure 11.4.

Geometrically, this is an intuitive choice and denotation. In Figure 11.5 the center is at $x = \frac{40+63}{2} = 51.5$, marked with the red dashed line. The search point (50, 5) is left of that line and therefore the *lo* child is chosen as the *near* child. Since the algorithm only considers the dimension used for the discriminator of the current node, it does not matter that the arrow pointing to *near* is actually longer than the arrow pointing to *far*.

Before recursing to *near*, the vectorized *FINDNEAREST* algorithm has an optimization opportunity not available to the scalar algorithm. If the search point is located between the discriminator values of the current node (such as for the root node in Figure 11.5), the current best distance might be less than Δ_{min}. In this case no record stored in or below *near* can yield a better distance and the recursion can be ended before reaching a leaf node. Since this can only be the case if a record in the current node yields the best distance, the current node's records must be considered before recursing. In Figure 11.5, the distance to the (60, 27) record in the root node thus yields the candidate with $distance = \sqrt{584}$.

Recursion to the *near* child invokes the *FINDNEAREST* algorithm on that node. The algorithm can make use of the current best distance to skip further recursion. Thus, it will not consider the *hi* child of *near* because it is designated as the *far* child (the search point is below the dashed line) and the current best distance from the root node is less than $\Delta_{max}^{(1)} = 77-5 = 72 > distance = \sqrt{584} \approx 23.2$. It must consider the records in the node itself, though, because $\Delta_{min}^{(1)} = 15 - 5 = 10 < distance$. The candidate from the *near* child in the example will be (30, 32) with $distance = \sqrt{1129}$, which is larger than the current best distance and therefore does not update the candidate.

If the *near* child is searched first, and thus the search point's discriminator value is less than or larger than all discriminator values of the current node, the algorithm subsequently checks whether records in the current node might have a shorter distance. However, if the current best distance is already less than Δ_{min}, neither the records in the node, nor the far child can yield a closer result and the algorithm can return. Geometrically, this is a test whether the hypersphere with radius *distance* around the search point reaches into the area of the current node. Otherwise, the candidate is updated if the current node contains a record with shorter distance.

After the *near* child and the current node have been considered, the algorithm might still need to descend the far child. In Figure 11.5, the circle around the search point with radius $\sqrt{584}$ reaches outside of the area of the root node. Thus, there could be a record closer to the search point in or below the far child. The algorithm determines this via comparing Δ_{max} (the maximum distance in the dimension of the discriminator to the records of the node) against *distance*.[9] If the far child needs to be considered, the algorithm recurses to that node to determine a candidate. To work efficiently, the recursion must always carry the current candidate for nearest neighbor to its child nodes. That way the recursion inside the children can use the current best distance value to avoid descending the tree unnecessarily. If the far child finds a better candidate it replaces the current best and returns it. At this point the node is done and can return to its caller.[10] In Figure 11.5, the far child determines (77, 2) with $distance = \sqrt{738}$ as its candidate, but discards it after comparing against the current best. The root node finally returns (60, 27) as the nearest neighbor of (50, 5).

9 At this point it is important that the distance of the far child in the dimension of the discriminator really corresponds to the distance of Δ_{max}. This would not be the case if the decision for *near* and far was made via a different method than the center between the minimum and maximum discriminator values of the node.

10 The call to the far child can thus use tail-recursion, which can be optimized as a jump instead of a full function call & return.

11.3.3 VC IMPLEMENTATION

The *INSERT* and *FINDNEAREST* algorithms can easily be implemented in C++ using the Vc library and especially the `simdize<T>` facility from Chapter 10. The source code for these algorithms is presented in Appendix E and the necessary data structures were already shown in Listing 11.7.

In addition to the presented `simdize<T>` facilities from Chapter 10, the code requires a `simdize_shifted` function. This function returns a new object where all vector entries are shifted to the left or right depending on the second argument.

The k-d tree implementations need three functions to determine the relevant properties from the objects inserted into the tree (`Point<float, N>` in the benchmark):

- `get_kdtree_distance` determines the distance between two points (or vectors thereof).
- `get_kdtree_1dim_distance` determines the distance between two points (or vectors thereof) in only one coordinate.
- `get_kdtree_value` returns the value of one coordinate to use as discriminator value (K_D).

11.3.3.1 OPTIMIZATIONS

The *FINDNEAREST* implementations were optimized for the target machine as much as possible. In a perfectly balanced tree with $2^n - 1$ nodes, there are 2^{n-1} leaf nodes and $2^{n-1} - 1$ nodes with two children. This is why all implementations first test whether the node is a leaf node case. In the vectorized implementation, the case of a search point between the minimum and maximum discriminator values of a node introduces one more case to special-case in the implementation.

In the vector implementations, the candidate object needs to be updated if a closer record is found in the current node. Since the candidate object is built as a vector (cf. Section 11.3.3.3), the update is executed as a masked assignment. The mask fully suffices to get correct results and thus no branching (`if` statement) is required. For larger types (point objects with more than 3 dimensions), the number of instructions needed to execute the masked assignment become expensive enough that an additional branch to skip the complete masked assignment is beneficial.

In all implementations, the tree is implemented in such a way that the dimension used for the discriminator is attached to the node type. This increases the code size but on the other hand allows the compiler to optimizes memory accesses and register usage more aggressively.

11.3.3.2 search key vectorization

As discussed in Section 11.2.1, there is a second approach for vectorization of the nearest neighbor lookup algorithm. To quantify the differences, I implemented a third *FINDNEAREST* function which searches for the nearest neighbors for multiple search keys in parallel. It reuses the scalar k-d tree data structure, and thus its *INSERT* implementation. The implementation for the `findNearest` function is presented in Listing E.5 (Appendix E).

11.3.3.3 candidate object

All three `findNearest` functions in the node classes use a single candidate object that is passed on from node to node to store the current best result and distance. In case of the vectorized nodes implementation, this candidate object stores a vector of objects and a vector of distances. This avoids horizontal reductions while traversing the tree. Only after the root node returns, a reduction is executed to determine the nearest neighbor.

11.3.4 benchmark

The benchmark program introduced in Section 11.2.4 compares the vectorized *FINDNEAREST* implementation against the equivalent scalar k-d tree implementation. The difference to Section 11.2.4 is that the program now examines the run times of inserts and nearest neighbor searches of the scalar and vectorized k-d tree data structures (**`KdTree`** and **`KdTreeV`**).

11.3.5 benchmark results & discussion

insert efficiency The run times of the scalar and vectorized *INSERT* algorithms (cf. Listing E.1 and Listing E.2 in Appendix E) are plotted in the first diagram of Figure 11.6. The second diagram shows the quotient of the two graphs from the first diagram. It shows that the vectorized implementation is not significantly slower than the scalar implementation, even though it requires a lot more code. To the contrary, the vectorized implementation is actually more efficient for larger trees. The ability of the vectorized *INSERT* algorithm to move objects that were already stored in a node to a child node leads to a more balanced tree than for the scalar k-d tree algorithm. With random input, the scalar k-d tree *INSERT* algorithm creates trees with a depth roughly 100%–150% larger than the optimal depth. The vectorized algorithm (with $\mathcal{W}_{\mathtt{T}} = 8$), creates trees with a depth roughly 0%–40% larger than the optimal depth. In both cases the tree is less balanced the more nodes are inserted.

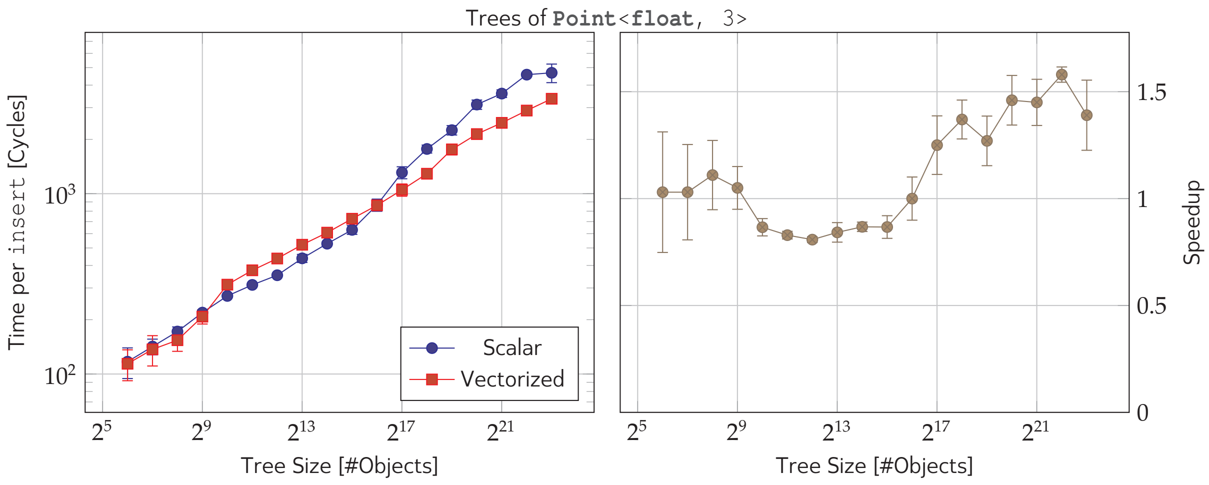


Figure 11.6: Benchmark results of k-d tree insertion. The left plot shows run times in CPU cycles. The right plot show the quotient and thus the speedup of the vectorized implementation over the scalar implementation.

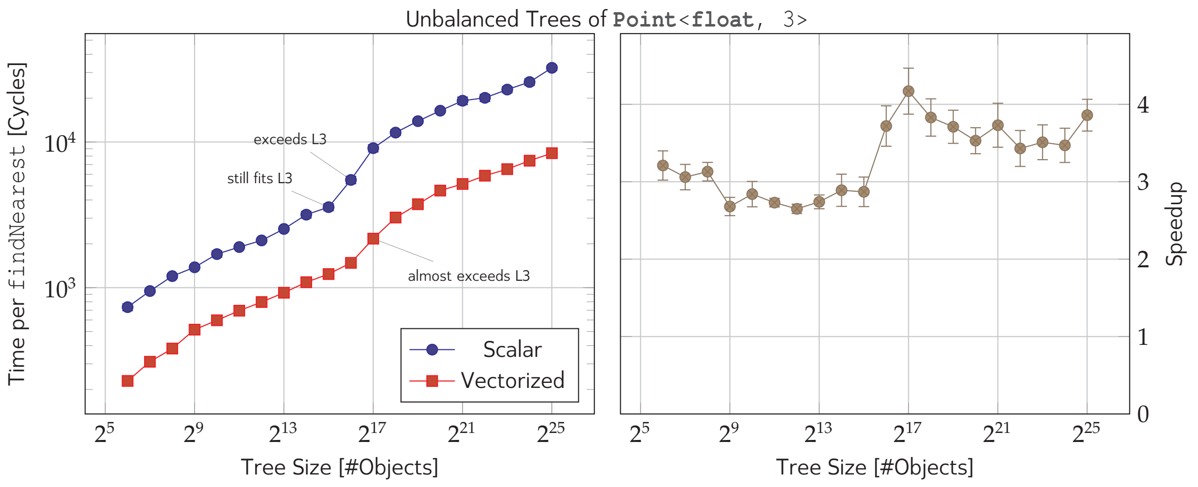


Figure 11.7: Benchmark results of k-d tree nearest neighbor searching. The left plot shows run times in CPU cycles. The right plot show the quotient and thus the speedup of the vectorized implementation over the scalar implementation.

FINDNEAREST EFFICIENCY The run times of the scalar and vectorized nearest neighbor search algorithms (cf. Appendix E) are plotted in the first diagram of Figure 11.7. The second diagram shows the quotient of the two graphs from the first diagram. It shows that the vectorized k-d tree nearest neighbor search is approximately a factor 3–4 faster. Since the tree is not rebalanced after random insertions, Figure 11.7 shows the combined efficiency of the *INSERT* and *FINDNEAREST* algorithms/implementations. If the tree is perfectly balanced when the *FINDNEAREST* algorithm is executed the scalar implementation executes roughly a factor four faster. This effect is less pronounced for the vectorized k-d tree implementation, which is why the relative speedup between scalar and vectorized k-d tree implementations decreases. Figure 11.8 shows the nearest neighbor search efficiency on balanced trees storing three-dimensional, six-dimensional, and nine-dimensional points.

HIGHER DIMENSIONS The k-d tree algorithm and its implementation (cf. Appendix E) allow an arbitrary number of dimensions. The effect of dimensionality on the efficiency of the *FINDNEAREST* algorithm was therefore tested with the benchmark and is shown in Figure 11.8. The scalar implementation exhibits approximately an order of magnitude difference in run time for three dimensional (—●—), six dimensional (—○—), and nine dimensional (—▲—) k-d tree nearest neighbor search. This scales slightly better for the vectorized implementation (—×—, —◆—, and —□—), which is clearly visible in the larger speedups for higher dimensions (—■—, —◇—, and —+—). This can be understood if we consider that the mean distance between the points in the random set increases with more dimensions. Consequently, the hypersphere around a candidate point on average has a larger radius and thus increases the likelihood for descending the *far* child. This is, of course, true for both the scalar and vector implementations. However, the additional tree traversal is more efficient through vectorization, therefore yielding the greater speedup.

MEMORY LATENCY The major cost of k-d tree nearest neighbor searching is the memory latency incurred by pointer chasing on tree traversal. The cost of a single node traversal is equivalent for the scalar and vector implementations. The important difference between the two implementations is the tree size. With $\mathcal{W}_{\mathtt{T}} = 8$ the vectorized tree has almost a factor of eight fewer nodes in the tree[11] and consequently the tree depth is three less. If we assume that the time spent at a single node in the tree is approximately equal for the scalar and vector implementations, then the vector implementation is only faster because it has to look at fewer nodes. The reduction of the tree depth is the major improvement of the vectorized implementation. A reduction of the tree depth is also possible with the scalar k-d tree (e.g. by storing multiple records in nodes and using an exhaustive nearest neigh-

11 Since the leaf nodes can be partially filled, the difference is not an exact factor of eight.

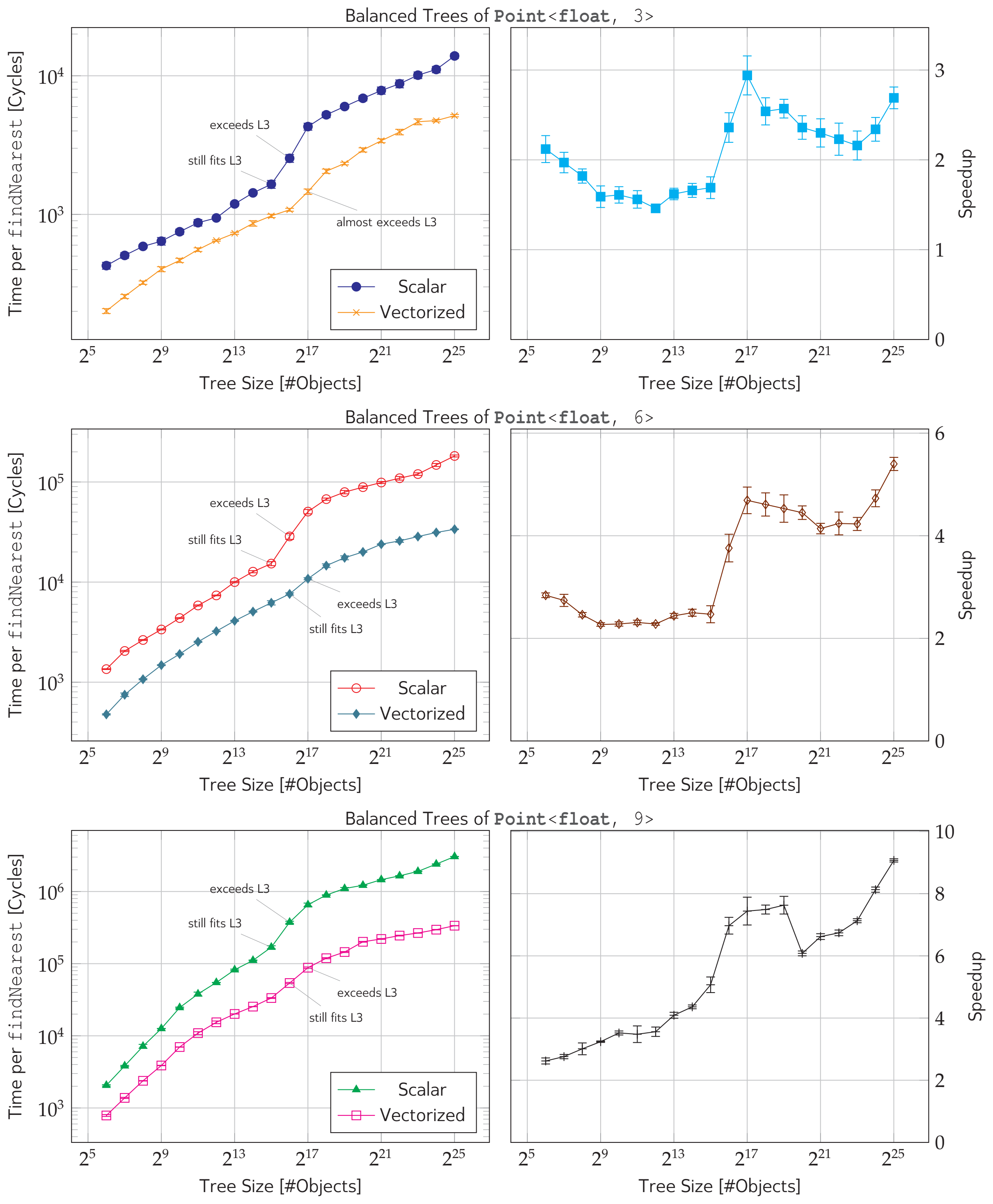


Figure 11.8: Benchmark of k-d tree lookup in three, six, and nine dimensions. The plots show the efficiency of lookups on balanced trees. (cf. Appendix E and Appendix C)

bor search for every node). This approach increases the time spent at a single node since it is basically a serialized implementation of the vectorized *FINDNEAREST* algorithm. Overall, the achievable speedup from comparison vectorization is diminished by the latency incurred from tree traversal. If tree traversal is considered as the serial part of the algorithm then Amdahl's law [3] describes the scalability issue of the parallelization of a search tree algorithm.

INFLUENCE OF CPU CACHES The graphs in the run time plots (left plot) of Figure 11.6, Figure 11.7, and Figure 11.8 follow $\mathcal{O}(\log N)$ complexity. At certain points (marked with "exceeds L3") the graphs bend upwards and continue with a different slope, though still with $\mathcal{O}(\log N)$ complexity. The points where the graphs bend upwards correspond to a haystack size where the complete k-d tree data structures occupies more memory than the level 3 cache can hold (3 MiB for the CPU that executed the benchmark). These points are different for the scalar and vectorized trees because the overhead from the tree structure is smaller for the vectorized tree. Every memory allocation (one per tree node) requires alignment to 16 Bytes and extra memory to store the length of the block [63]. Therefore, each node in the scalar k-d tree requires 48 Bytes of memory to store one record.[12] Each node in the vectorized k-d tree (in this case with AVX and `Point<float, 3>`) requires 160 Bytes of memory to store one node and thus only 20 Bytes per single record.[13]

11.3.5.1 VECTORIZATION DIRECTION

Figure 11.9 plots the run time normalized to the number of search keys for calls to `findNearest` of the three implementations in Appendix E. It is clearly visible that vectorization of the search key (—●—) leads to the most efficient k-d tree implementation for small trees. From 40 or more entries in the tree, the implementation using node vectorization (—■—) wins over search key vectorization. From ca. 120 or more entries, even the scalar implementation (—●—) outperforms search key vectorization. The graph for vectorization on the search key shows linear scaling on N ($\mathcal{O}(N)$ complexity). This implies that on average the majority of the tree nodes needs to be searched. Consider that the search algorithm has to find eight (the benchmark was compiled for AVX with $\mathcal{W}_{\texttt{float}} = 8$) random records in the tree. Thus, it is obvious that for most of the time the largest distance of the candidate set will lead to traversal of the *far* child and consequently lead to traversal of almost the complete tree structure.

If search key vectorization leads to a mostly exhaustive search, then comparing against the exhaustive search implementation from Section 11.2.2 is inevitable. The

12 16 Bytes for the two child-pointers. 12 Bytes for the `Point` object; padded to 16 Bytes. 16 Bytes for malloc bookkeeping.

13 16 Bytes for the two child-pointers; padded to 32 Bytes. 96 Bytes for the `simdize<Point>` object. 16 Bytes for malloc bookkeeping; padded to 32 Bytes.

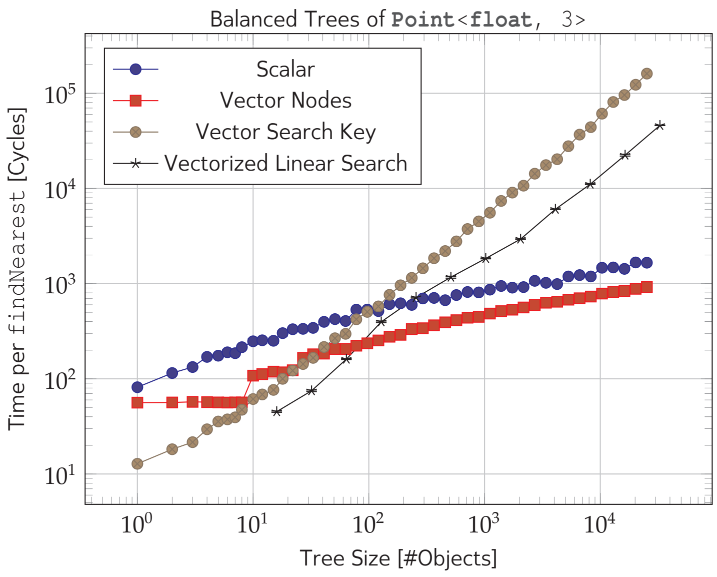


Figure 11.9: k-d tree vectorization over the search key run times.

results from the exhaustive nearest neighbor search are plotted as a fourth graph (—⋆—) in Figure 11.9. The efficiency is consistently better for the simple linear implementation. Note that the linear search uses scalar search keys and returns values, though. This makes it perform worse at very small haystacks (especially below $\mathcal{W}_{\mathtt{T}}$) since the conversion between **T** and **simdize**<**T**> adds overhead which is not present with (linear) search key vectorization.

Note the plateau of the vectorized nodes implementation (—■—) at tree sizes 1–8. This is due to the root node being able to store 8 points ($\mathcal{W}_{\mathtt{float}} = 8$). Consequently, the run time of a nearest neighbor search is equal for 1–8 entries in the tree. A second, though slightly inclined plateau is visible in the range 9–24, where the depth of the k-d tree is two.

11.4 CONCLUSION

There exist different variants of the k-d tree data structure in the literature and in applications. It will be interesting to apply the vectorization approach to these variants as well. Preliminary tests have shown additional potential in vectorization of search trees with additional reduction of the depth (e.g. with $\mathcal{W}_{\mathtt{T}} + 1$ instead of 2 child nodes).

The work of this chapter has shown that vectorization of search data structures is possible with vector types; allowing explicit expression of data-parallel storage and execution. This is a major improvement over loop-vectorization, which only solves the programming language shortcoming of expressing data-parallel execution. Especially useful in the development of the vectorized search algorithms and data structures was that vector types guide the developer to the solution by suggesting storage via vector types, which in turn necessitates improvements to the *INSERT* and *FINDNEAREST* algorithms.

12 VC IN HIGH ENERGY PHYSICS

The track reconstruction software for the HLT[1] [2, 13, 19] at ALICE[2] [12, 14], an experiment at the LHC[3] [9] in Geneva, was the original application for Vc [55]. Rohr [72] describes how, since then, the track reconstruction software as been optimized for GPU usage and relies on OpenCL [65] and CUDA [15] for parallelization. For CPU development, Vc has become an important building block in High-Energy Physics software and is thus in production use on thousands of different machines with different operating systems and C++ compilers.

The following sections will describe some of these projects. The first example on ALICE analyzes the use of Vc and the vectorization approach in depth.

12.1 ALICE

The ALICE detectors record information about particles that pass through it. It uses many different subdetectors to record different information and for different particle types. The central detectors for tracking the trajectories of charged particles are the TPC[4] and ITS[5] detectors. These detectors record space points where a particle passed through and interacted with the material/gas at this point. The detectors cannot relate the space points to individual particles. However, the reconstruction algorithms can recover this relation. Particle identification and energy measurement is possible with different subdetectors on the outer layers.

The task of the ALICE software is the reconstruction of the physical events in the detector from the measurements of the detectors. The track reconstruction software analyzes space points to reconstruct the trajectories of charged particles. In principle, this is just a complex “connect the dots” task. Track reconstruction yields information about the momentum, primary and secondary vertexes (origins for several trajectories), and the expected trajectory through the outer detectors.

1 High-Level Trigger
2 A Large Ion Collider Experiment
3 Large Hadron Collider
4 Time Projection Chamber
5 Inner Tracking System

Finally, the complete physics information yields a description of the events in the detector. The main interest in ALICE is the research of matter at high energies and densities. Since the detector can not measure the events in the vacuum at the center of the detector, the reconstructed track information is used to infer the physics at the collision points.

12.1.1 APPROXIMATING MAPS

Some of the reconstruction algorithms require the transformation of the measured quantities to error-corrected physical quantities. The issue is that not all measurements can be mapped directly with an analytical function to known reference values. These corrections are measured via detector calibration.

One example of such an adjustment is the transformation of pad/time coordinates (TPC coordinates) to three-dimensional space points (x, y, z). In principle, the pads at the end-caps of the TPC determine the x and y coordinates and the drift times in the TPC infer the z coordinate. However, several corrections need to be applied, correcting for small variations in alignment of the detector parts, space charge in the TPC, and other distortions. The TPC is therefore calibrated via matching trajectories from the ITS subdetector to trajectories from the TPC. The result is that several correction steps are applied in the ALICE software when converting from pad/time coordinates to space points. These corrections are implemented in the ALICE Offline software stack and require a lot of conditional execution and memory lookups in tabulated correction data.

This implementation of coordinate transformation is inefficient, though, because it requires several memory lookups in large data structures and a lot of branching. Sergey Gorbunov therefore approximated the transformation via a spline implementation. The spline calculation is still computationally expensive. Vectorization of the calculation can speed up the spline implementation further, thus making this critical transformation as efficient as possible.

12.1.2 SPLINE VECTORIZATION

The current ALICE code base using this transformation does not employ horizontal vectorization, where a vector of pad/time inputs would yield a vector of three-dimensional space-points. Additionally, horizontal vectorization requires gathers in the lookup of the tabulated values for the spline, since the tabulated values are different for different pad/time input values. As the following sections will show, the efficiency of the vectorized implementation is mainly limited by load throughput of the tabulated values and thus horizontal vectorization does not easily improve over vertical vectorization.

```cpp
template <typename T> static inline T GetSpline3(T v0, T v1, T v2, T v3, T x) {
  const T dv = v2 - v1;
  const T z0 = T(0.5f) * (v2 - v0);
  const T z1 = T(0.5f) * (v3 - v1);
  return (x * x) * ((z1 - dv) * (x - 1) + (z0 - dv) * (x - 2)) + (z0 * x + v1);
}
```

Listing 12.1: Implementation of an elementary spline.

Vertical vectorization is still interesting for this task since 15 elementary splines (cf. Listing 12.1) have to be calculated for implementing the 2D input / 3D output approximation. Five elementary splines are required per dimension in the output (cf. Figure 12.1): Four splines in the second input dimension[6] calculate the input data for the fifth spline in the first input dimension. Thus, there are twelve independent splines calculated from the tabulated data which are used as input for the three subsequent splines. Consequently $12 \cdot 4 = 48$ tabulated values must be loaded from memory for the complete coordinate transformation.

The spline function in Listing 12.1 is a generic implementation supporting both fundamental arithmetic types as well as any of the Vc vector types. This makes it easy to build and test different implementations using different memory layouts with a single spline function.

12.1.2.1 IMPLEMENTATION IN PRODUCTION USE

The implementation in production use at ALICE uses vertical vectorization. The tabulated values build a two-dimensional table (for the two input dimensions) with three values in each cell (for the three output dimensions). This is depicted in Figure 12.1. The dashed magenta arrow (- ->) and the dotted green arrow (·····>) point out the two principal memory orders for storing the tabulated values. Since each $\tau_{i,j} = (x_{i,j}, y_{i,j}, z_{i,j})$ stores three values there are three natural options:

1. Store three separate tables for $x_{i,j}$, $y_{i,j}$, and $z_{i,j}$.
2. Store the complete $\tau_{i,j} = (x_{i,j}, y_{i,j}, z_{i,j})$ in each point of the table.
3. Pad $\tau_{i,j}$ with an additional zero and store it in each point of the table.[7]

In the ALICE implementation the memory order follows the dotted green arrow (·····>) and uses variant 3 for storage of τ. The vectorization is chosen over τ, which is the reason for the zero-padding. Thus, a single vector load for one table cell at (i, j) loads $(x_{i,j}, y_{i,j}, z_{i,j}, 0)$. These vectors are subsequently used as input variables to the elementary spline function (cf. Listing 12.1).

6 These four splines may be evaluated in arbitrary order.
7 This is a typical approach for vertical vectorization, in order to allow aligned vector loads for three-dimensional data.

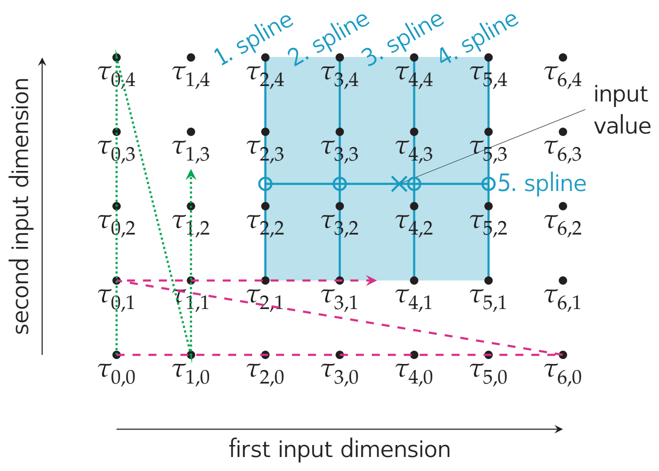

1. spline
2. spline
3. spline
4. spline
5. spline
input value
second input dimension
first input dimension
$\tau_{0,4}$ $\tau_{1,4}$ $\tau_{2,4}$ $\tau_{3,4}$ $\tau_{4,4}$ $\tau_{5,4}$ $\tau_{6,4}$
$\tau_{0,3}$ $\tau_{1,3}$ $\tau_{2,3}$ $\tau_{3,3}$ $\tau_{4,3}$ $\tau_{5,3}$ $\tau_{6,3}$
$\tau_{0,2}$ $\tau_{1,2}$ $\tau_{2,2}$ $\tau_{3,2}$ $\tau_{4,2}$ $\tau_{5,2}$ $\tau_{6,2}$
$\tau_{0,1}$ $\tau_{1,1}$ $\tau_{2,1}$ $\tau_{3,1}$ $\tau_{4,1}$ $\tau_{5,1}$ $\tau_{6,1}$
$\tau_{0,0}$ $\tau_{1,0}$ $\tau_{2,0}$ $\tau_{3,0}$ $\tau_{4,0}$ $\tau_{5,0}$ $\tau_{6,0}$

Figure 12.1: Layout and usage of the tabulated values.

This vectorization approach requires $\mathcal{W}_{\texttt{float}}$ to be exactly four. The ALICE implementation thus simply fails to vectorize if $\mathcal{W}_{\texttt{float}} \neq 4$. This solution obviously leaves room for improvement.

As Section 12.1.3.1 will show, the speedup of this implementation is roughly a factor two, but only on systems with $\mathcal{W}_{\texttt{float}} = 4$. Therefore, the following will discuss alternative implementations. After benchmarking the different approaches the results will be discussed.

12.1.2.2 ALTERNATIVE IMPLEMENTATION USING ADVANCED FEATURES

The following discussion uses these type aliases:

```
typedef Vc::SimdArray<float,  4> float4;
typedef Vc::SimdArray<float, 12> float12;
typedef Vc::SimdArray<float, 16> float16;
```

Float4 This implementation is a slight modification of the original ALICE implementation (cf. Listing F.9). It uses the **`float4`** type instead of **`float_v`** to solve the portability issue mentioned above. This allows the formulation of the algorithm in a way which expresses the fixed width of the data-parallelism in the algorithm. This is an example for the use-case mentioned in Section 7.3.2. It uses (aligned) vector loads to load $\{x, y, z, 0\}$ into 4-wide vectors for a threefold parallelization of the spline calculation.

The Float4 implementation is shown in Listing F.3.

Float16 In the Float4 implementation, the 12 initial splines are executed as four vectorized spline evaluations using $\{x, y, z, 0\}$ as inputs. This can be combined into a single spline evaluation using **`float16`**. The Float16 implementation therefore combines four `float4` loads via `simd_cast` to a single `float16` object. The `float16` object resulting from the spline evaluation is subsequently partitioned into four `float4` objects again to evaluate the final three splines.

The FLOAT16 implementation is shown in Listing F.4.

FLOAT12 By changing the storage order of the tabulated data to follow the dashed magenta arrow (-->) in Figure 12.1 and variant 1 for τ, it is possible to do vector loads with four relevant values. Thus, the four vector entries with value 0 in the FLOAT16 implementation are optimized out. Instead of four vectors with 75% relevant data, the implementation only loads three vectors. Each `float4` object then contains 100% relevant data. The `float12` type can be implemented using one eight-wide vector and one four-wide vector on a system with $\mathcal{W}_{\texttt{float}} = 8$.

The FLOAT12 implementation is shown in Listing F.5.

FLOAT12 INTERLEAVED The third storage order uses variant 2 for τ and follows the dashed magenta arrow (-->) in Figure 12.1. Four `float12` vector loads thus yield the $\{x, y, z\}$ inputs for all twelve initial splines. The resulting vector with $\mathcal{W}_{\texttt{float}} = 12$ is subsequently dissected into four `float4` objects storing the four $\{x, y, z\}$ values and an arbitrary fourth value for the final three splines.[8]

The FLOAT12 INTERLEAVED implementation is shown in Listing F.6.

HORIZONTAL 1 In order to investigate the potential of a different vectorization direction—which requires vectorization of the calling code—I implemented a horizontal vectorization of the approximation. A vector of two-dimensional inputs with native vector width ($\mathcal{W}_{\texttt{float}}$) is used as input to the function. The HORIZONTAL 1 implementation then executes $\mathcal{W}_{\texttt{float}}$ times more splines with a single call than the implementations above. In addition to vectorizing the splines, a horizontal implementation also vectorizes the calculation of the indexes for the tabulated data.

This implementation uses the data structures from the FLOAT4 implementation. Since, independent from the direction over τ in Figure 12.1, there are always four $\tau_{i,j}$ elements consecutively in memory and each $\tau_{i,j}$ stores four scalar values, the implementation could load up to sixteen consecutive scalar values with a vector load. However, the loaded values need to be transposed into native vector objects. Therefore it is not beneficial to load all values at once. The consecutive loads with subsequent transposition implement

8 In principle, this should use `SimdArray<float, 3>`, but as this is harder to optimize for an implementation (and currently not optimized in Vc), I chose `float4`.

a gather operation, which can execute more efficiently because it makes use of the data-locality built into the data structure. The expression `Vc::tie(x, y, z) = map[ind]` is implemented in Vc as a vector load which loads three consecutive values at the memory addresses identified by `map[ind[0]]`, `map[ind[1]]`, `map[ind[2]]`, `map[ind[3]]`, … and subsequently uses vector instructions for transposition. The vector transposition places the first values of each load into `x`, the next into `y`, and the final values into `z`.

The Horizontal 1 implementation is shown in Listing F.7.

Horizontal 2 This implementation uses the same interface as the Horizontal 1 implementation. However, the storage order for the tabulated values is different. It equals the order from Float12 (separate tables for x, y, and z and neighboring values in the first input dimension). Thus, it can load and deinterleave four vectors per gather: `Vc::tie(x[0][0], x[1][0], x[2][0], x[3][0]) = fXYZ[ind]`. After loading 16 vectors (four gathers), all spline calculations for a single output dimension can be executed. This is therefore repeated twice for the y and z results.

The Horizontal 2 implementation is shown in Listing F.8.

An important consequence of using `SimdArray<T, N>` for vertical vectorization is the ability of using the best available vector instructions on systems with $\mathcal{W}_{\texttt{float}} > 4$. For example, on an AVX system, the Float16 implementation will execute two elemental spline evaluations using `float_v` objects with eight entries each. On an SSE system it compiles to four elemental spline evaluations instead.

12.1.3 BENCHMARK

To evaluate the efficiency of the different spline implementations I adjusted the benchmark program from Chapter 11. It constructs the splines in such a way that they can approximate the transformation $[-1,1]^2 \to [-1,1]^3$. The number of tabulated values for the spline is increased starting from 4^2 up to 235^2. This tests the complexity of the algorithm (which is supposed to be $\mathcal{O}(1)$) and shows how the CPU caches influences the run time.

Each run time measurement is repeated (and the measurement discarded) until the standard deviation of the 100 timed executions is below 5%. This helps eliminating results that were influenced by a busy system which would increase the deviations in consecutive executions.

The tabulated values for the different spline data structures are initialized with the exact same values. This makes the run times comparable and allows verification that the results are equivalent. For comparability, the spline evaluations are also executed with the same input coordinates for each run time measurement.

Starting with GCC version 4.9, the compiler is able to auto-vectorize the scalar implementation. Since it is easier to reason about a scalar implementation that is not auto-vectorized and since ALICE does not use GCC 4.9 in production use yet, auto-vectorization is turned off for the benchmark. For comparing manual vectorization against automatic vectorization, the scalar implementation is compiled and benchmarked a second time with auto-vectorization turned on.

Refer to Appendix F Listing F.11 for the complete benchmark code.

12.1.3.1 results

Figures 12.2 & 12.3 show the results of the benchmark compiled with the AVX or SSE implementations of Vc, respectively. The benchmark was executed on an Intel i5 3360M CPU with 2.80 GHz ("Turbo Mode" and power management were disabled—cf. Appendix D—). The code was compiled with GCC version 4.9.1 using the following optimization relevant compiler flags:

```
-O3 -ffp-contract=fast -march=core-avx-i -DNDEBUG.
```

The plots on top of Figures 12.2 & 12.3 show the run time of the coordinate transformation approximation normalized to a single transformation.[9] The run time of the Scalar (—■—) implementation is compared against all the other implementations in the "Speedup" plots below.

All graphs, except for the Alice (—◇—) graph, exhibit an approximately flat scaling behavior ($\mathcal{O}(N)$) with respect to the size of the map. At about ca. $2 \cdot 10^3$ tabulated values, which corresponds to the size of the L1 cache of 32 KiB, the graphs bend upwards. There is a second bend at ca. $1.5 \cdot 10^4$ tabulated values, corresponding to the size of the L2 cache of 256 KiB. The Alice (—◇—) implementation runs longer for small maps and slightly better for larger map sizes (compare Alice (—◇—) against Scalar (—■—) in Figure 12.2 and Alice (—◇—) against Float4 (—●—) in Figure 12.3). This is discussed in Section 12.1.3.2.

In Figures 12.2 & 12.3, the run times of Float4 (—●—), Float16 (—+—), and Float12 Interleaved (—□—) are approximately equal. For AVX (Figure 12.2), the efficiency of Float16 (—+—) is slightly higher compared to the other two, though. The Float4 (—●—) and Scalar (—■—) implementations show the same run times in the AVX and SSE variants. The Float12 (—○—) implementation is slightly slower than Float4 (—●—), with the efficiency degrading faster for larger map sizes.

The Horizontal 1 (—×—) and Horizontal 2 (—◆—) implementations exhibit the shortest run times in Figure 12.2 and results comparable to Float4 (—●—) or Float12 (—○—) in Figure 12.3. Note that the Horizontal 2 (—◆—) implementation

9 Note that the Horizontal 1 (—×—) and Horizontal 2 (—◆—) implementations evaluate $\mathcal{W}_{\texttt{float}}$ transformations with a single function call.

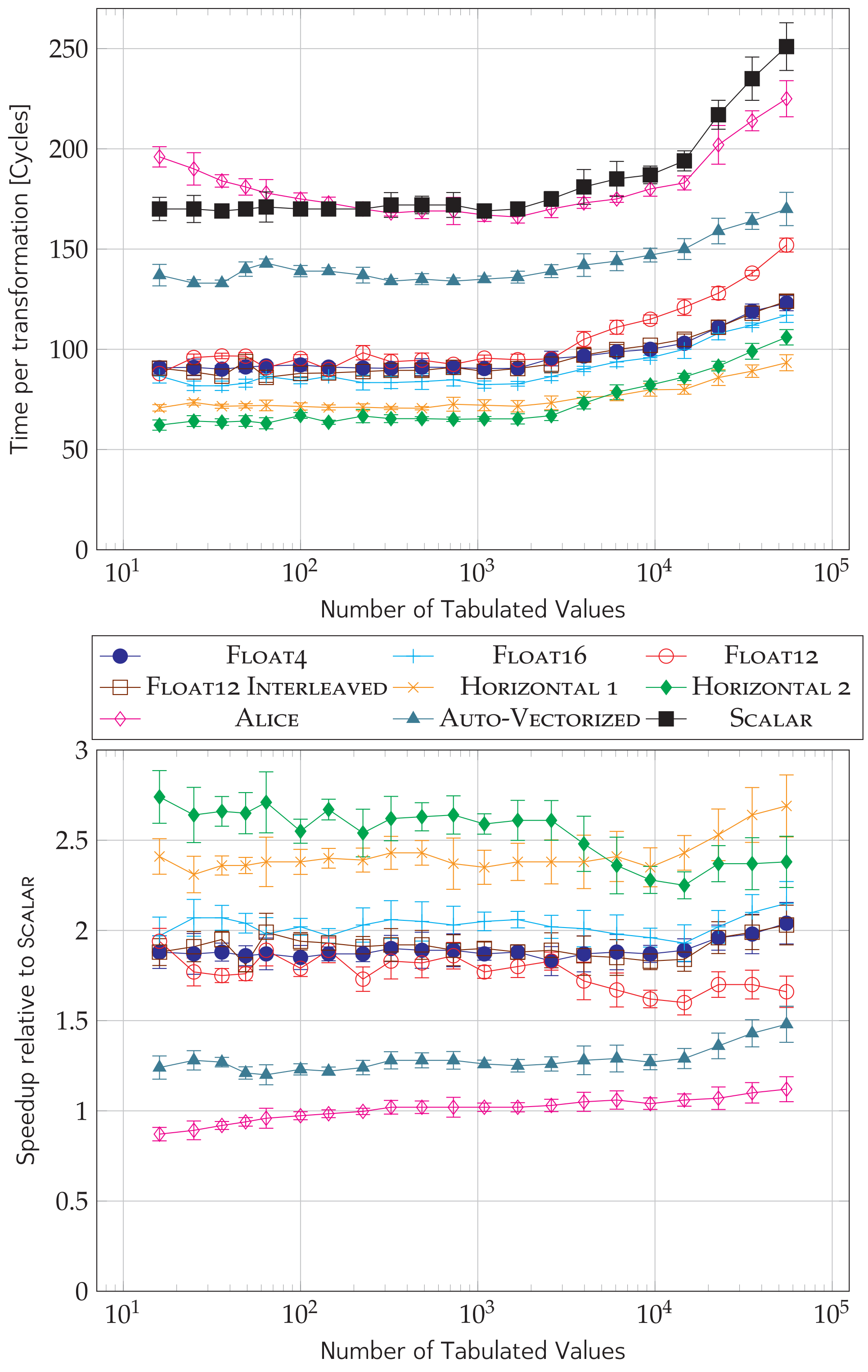


Figure 12.2: Efficiency of the spline implementations using Vc's AVX implementation.

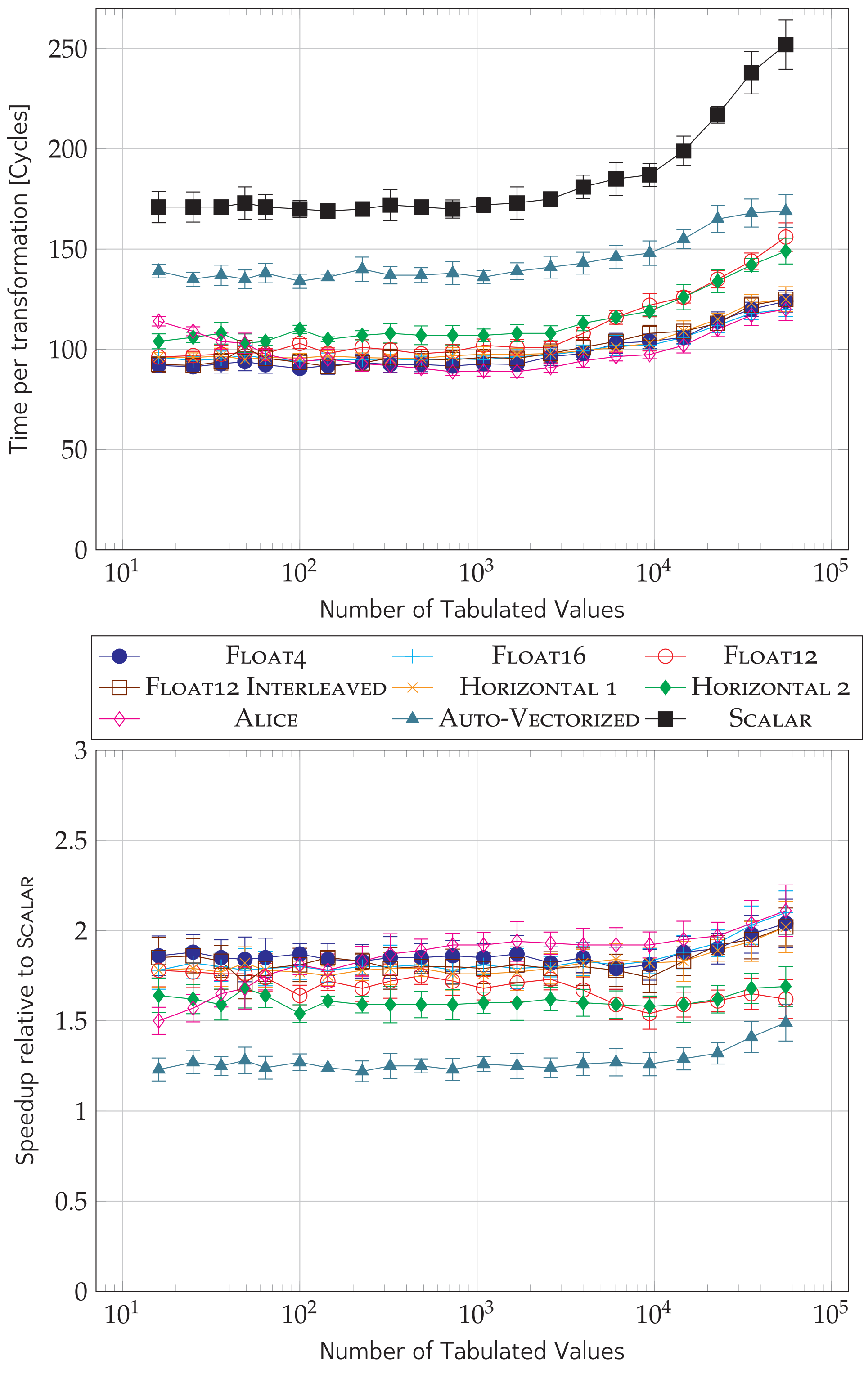


Figure 12.3: Efficiency of the spline implementations using Vc's SSE implementation.

is more sensitive to larger map sizes than the Horizontal 1 (—×—) implementation.

Finally, the Auto-Vectorized (—▲—) graph clearly shows that the auto-vectorizer improved the run time of the Scalar (—■—) implementation. Nevertheless, it runs longer than any of the explicitly vectorized implementations for AVX and SSE.

12.1.3.2 discussion

alice The behavior of the original Alice (—◇—) implementation is counterintuitive. The expectation is that the efficiency degrades only due to memory loads which miss the cache (i.e. for larger map sizes). Investigation of the implementation has shown that this is due to the use of `if` statements for the calculation of the memory addresses (cf. Listing F.9). This is translated by GCC to jump instructions and thus leads to branching and mispredictions on execution. The branches are more expensive for smaller maps, where branch prediction is more likely to be incorrect. If the implementation instead uses `min`/`max` instructions to clamp the index values (cf. Listing F.2), the efficiency only depends on cache misses and consequently only starts degrading for map sizes that exceed the respective cache sizes. Since all the other implementations use `min`/`max` instructions for clamping, they show a flat graph up to the point where the L1 cache is exceeded. The Alice (—◇—) implementation is more efficient for larger map sizes because the CPU can speculate that no index clamping is necessary. It can thus fetch data from memory earlier than the implementations which first have to wait for the results of the `min` and `max` instructions.

The Alice (—◇—) implementation is written using **`float_v`** directly, and therefore falls back to the scalar implementation if $\mathcal{W}_{\texttt{float}} \neq 4$. This is why the Alice (—◇—) graph in Figure 12.2 is on the order of the Scalar (—■—) implementation.

attainable speedup / amdahl's law In general, the speedup cannot attain the full vector width. For the vertically vectorized implementations this is mostly due to the latency induced by offset calculation into the map of tabulated values as well as the subsequent load latencies. The Float12 (—⊖—) implementation requires vector transposition and therefore introduces additional latency between the actual parallelized spline evaluation. The horizontal vectorizations require deinterleaving gathers and therefore also introduce additional latency before the actual parallelized spline evaluation. According to Amdahl's law [3], this limits the attainable speedup and especially limits scalability with regard to the vector width.

horizontal implementations The vectorization opportunities for horizontal vectorization are better because in addition to parallelizing the splines, the index calculation can be executed in parallel. On the one hand, this solves the Amdahl problem since the horizontal vectorization parallelizes the complete execution of the approximation function. On the other hand, a different map index per vector entry implies the need for gather operations, which load the input arguments to the elemental spline functions. Since gather operations are, in general, more expensive than loads of successive values in memory this limits the attainable speedup for horizontal vectorization. Figure 12.3 shows that the horizontal implementations do not achieve higher efficiencies than vertical vectorization. This is different for AVX; Figure 12.2 shows that horizontal vectorization scales better with the vector width for this problem.

index type While testing the influence of the index calculation strategy on the total efficiency of each approximation implementation, I noticed that there is a 10% difference in efficiency depending on whether the index variable is declared as a signed or unsigned integer. Using `unsigned int` the total run time is longer than with `int`. This was not investigated further and all implementations simply use `int` for the benchmark. The discrepancy is likely due to the different overflow properties of `unsigned int` and `int`. An `unsigned int` is specified to implement modulo 2^n arithmetics. For an `int`, on the other hand, it is undefined behavior if the variable over- or underflows. Subsequently, the compiler can assume that adding a positive value to an `int` variable yields a larger value. This is not true for `unsigned int` and thus inhibits certain optimizations in combination with pointer arithmetics.

speedup with vertical vectorization The Float16 (—+—) and Float4 (—●—) efficiencies for SSE/AVX are expected. The benchmark result shows how, on the one hand, vertical vectorization, using 3-component vectors, can also benefit from wider vectors if an additional dimension of data-parallelism exists in the algorithm. The Float4 (—●—) implementation, on the other hand, shows that with x86, the code cannot be translated to a more efficient variant, but at the same time also does not have to fall back to scalar execution, as in the Alice (—◇—) implementation. Using a larger `SimdArray<float, N>` type enables the solution to scale slightly better to targets up to $\mathcal{W}_{\texttt{float}} = 16$. In this problem this is limited to a single vectorized spline evaluation which can use the `float16` type. The spline evaluation in the second dimension only requires three spline evaluations and therefore cannot scale above $\mathcal{W}_{\texttt{float}} = 4$. The Float12 (—○—) and Float12 Interleaved (—□—) implementations avoid the calculation of 4 unused spline evaluations present in the Float4 (—●—) and Float16 (—+—) implementations. However, this requires a different memory layout for the map which leads to unaligned loads. Subsequently, many of the loads span over two cache lines, further increasing the latency introduced by the vector gather/deinterleave code. This increases the serial portion of the code and limits the attainable speedup.

AUTO-VECTORIZATION The efficiency of AUTO-VECTORIZED (–▲–) must be compared against FLOAT4 (–●–). The latter is the closest equivalent for the vectorization approach the compiler takes. It shows that auto-vectorization can only make limited assumptions about the code and thus not transform it into the clear data-parallel execution possible via explicit vectorization.

12.2 KALMAN-FILTER

The Kalman-filter [51] is an important tool in applications that need to predict a trajectory or need to find a set of parameters (and their errors and correlations) that describe a trajectory. As such, it is used extensively in track reconstruction algorithms for high energy physics experiments (cf. [29]). In these experiments, up to several thousand particles move from the collision point through the detector and its strong magnetic field (cf. Section 12.1). The detector itself contains material which interacts with the measured particles. Therefore, the magnetic field, the detector material, and the properties of the particle determine the trajectory through the detector.

12.2.1 VECTORIZATION

Vectorization of the Kalman-filter has been researched and implemented by Gorbunov et al. [33, 32], Gorbunov et al. [34], Kisel et al. [53], and Kretz [55] for many different target systems. Kretz [55] developed the Vc abstraction as part of porting the ALICE related work to the Intel Larrabee.

Based on this previous work, the track reconstruction software for ALICE (at CERN[10]), STAR[11] (at RHIC[12]), and CBM[13] (at FAIR[14]) make use of vectorized Kalman-filter implementations. While ALICE turned to GPUs and currently uses a CUDA/OpenCL implementation[15], the Vc based ALICE tracker has been adapted by Fisyak et al. [23] for the STAR experiment. At this point the STAR experiment uses Vc in production code in both online and offline software (J. Lauret, personal communication, 02/20/2015).

10 European Organization for Nuclear Research; in Geneva
11 Solenoidal Tracker at RHIC
12 Relativistic Heavy Ion Collider; at Brookhaven National Laboratory
13 Compressed Baryonic Matter; experiment at FAIR
14 Facility for Antiproton and Ion Research; in Darmstadt
15 The target hardware for the Vc implementation did not make it to market, necessitating the use of NVIDIA or AMD GPUs.

Kisel et al. [52] developed the *KF Particle* package and recently vectorized the project using the Vc library (M. Zyzak, personal communication, 02/20/2015). This work enables fully vectorized usage of Xeon Phi accelerator cards in addition to full utilization of the host CPUs.

12.2.2 EXAMPLE

Listing 12.2 shows a (slightly reduced) Kalman-filter example from track reconstruction. It is used to fit the track parameters of several particles in parallel. In the `Filter` function a new measurement (`m`) is added to the **`Track`** state. The data structures in the example are declared as structures of vectors (cf. *AoVS* in Chapter 10). The objects thus each contain the data of $\mathcal{W}_{\texttt{float}}$ tracks. Consequently, instead of working with one track at a time, the code explicitly states that multiple tracks can be filtered in parallel and that their data is stored interleaved in memory.

12.3 GEANT-V

GEANT-V[16] is a “particle transport application used in detector simulation” [10]. The preceding GEANT versions are widely used in high energy physics software. Improving its performance will therefore benefit the whole community. Carminati et al. [10] write:

> Monte Carlo simulation is one of the most demanding computing task, due to the slow decrease of the statistical fluctuations around the estimated mean, which is proportional to the inverse of the square root of the number of events simulated. Moreover particle transport simulation is one of the most experiment-independent applications in High Energy Physics.

It is therefore a very important result if this software can execute more efficiently. The computing costs for high energy physics experiments could be significantly reduced.

12.3.1 DATA-PARALLEL PROCESSING

According to Carminati et al. [11], the particle transport problem the GEANT[17] software addresses contains a lot of intrinsic parallelism. There are many events that need to be processed according to the same rules. Inside these events are tracks which again need to be processed in the same way. Besides multi-core optimizations, the GEANT-V project in particular investigates vectorized processing.

16 GEometry ANd Tracking Vector (Prototype) project
17 GEometry ANd Tracking

```cpp
struct Covariance {
  float_v C00,
          C10, C11,
          C20, C21, C22,
          C30, C31, C32, C33,
          C40, C41, C42, C43, C44;
};

struct Track {
  float_v x, y, tx, ty, qp, z;
  float_v chi2;
  Covariance C;

  float_v NDF;
  // ...
};

struct HitInfo {
  float_v cos_phi, sin_phi, sigma2, sigma216;
};

void Filter(Track &track, const HitInfo &info, float_v m) {
  Covariance &C = track.C;
  const float_v residual =
      info.cos_phi * track.x + info.sin_phi * track.y - m;          // ζ = Hr - m
  const float_v F0 = info.cos_phi * C.C00 + info.sin_phi * C.C10;  //  CH
  const float_v F1 = info.cos_phi * C.C10 + info.sin_phi * C.C11;
  const float_v F2 = info.cos_phi * C.C20 + info.sin_phi * C.C21;
  const float_v F3 = info.cos_phi * C.C30 + info.sin_phi * C.C31;
  const float_v F4 = info.cos_phi * C.C40 + info.sin_phi * C.C41;
  const float_v HCH = F0 * info.cos_phi + F1 * info.sin_phi;        // HCH
  const float_v wi = 1.f / (info.sigma2 + HCH);
  const float_v zetawi = residual * wi;  // (V + HCH)^1 ζ
  const float_v K0 = F0 * wi;
  const float_v K1 = F1 * wi;
  const float_v K2 = F2 * wi;
  const float_v K3 = F3 * wi;
  const float_v K4 = F4 * wi;
  track. x -= F0 * zetawi;               // r -= CH (V + HCH)^1 ζ
  track. y -= F1 * zetawi;
  track.tx -= F2 * zetawi;
  track.ty -= F3 * zetawi;
  track.qp -= F4 * zetawi;
  C.C00 -= K0 * F0;                      // C -= CH (V + HCH)^1 HC
  C.C10 -= K1 * F0;
  C.C11 -= K1 * F1;
  C.C20 -= K2 * F0;
  C.C21 -= K2 * F1;
  C.C22 -= K2 * F2;
  C.C30 -= K3 * F0;
  C.C31 -= K3 * F1;
  C.C32 -= K3 * F2;
  C.C33 -= K3 * F3;
  C.C40 -= K4 * F0;
  C.C41 -= K4 * F1;
  C.C42 -= K4 * F2;
  C.C43 -= K4 * F3;
  C.C44 -= K4 * F4;
  track.chi2 += residual * zetawi;       // χ^2 += ζ (V + HCH)^1 ζ
  track.NDF += 1;
}
```

Listing 12.2: Example Kalman-filter implementation from the SIMD-KF benchmark.

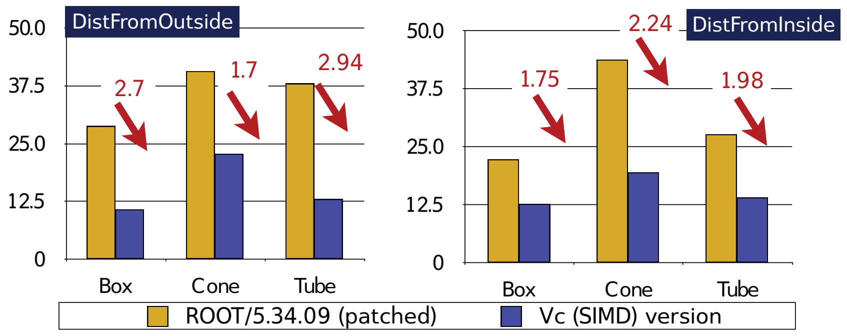


Figure 12.4: Speedups obtained from vectorising simple algorithms for the box, cone and tube shapes from the ROOT shape library. The functions presented are `distFromOutside` and `distFromInside`, i.e. the distance to the shape boundary from a point outside and from inside the shape respectively. [reproduced from 84]

Wenzel et al. [84] then discuss the details of the vectorization effort, which was initiated "on the geometry component":

> The necessary prerequisite is the availability of (contiguous) data on which the same operations should be carried out. In the Geant-Vector prototype, this is being realised by grouping the particles in the same logical volume (potentially from different events) into a data-parallel container called a basket [31].

12.3.2 USES OF VC

Wenzel et al. [84] write about the choices they considered for vectorization and their experience with Vc:

> Considering the difficulties of trying to get the code to autovectorise, versus the relative ease of programming (and compiler independence) with a library like Vc, we then opted for the second choice for the purpose of this first performance evaluation. At the time of writing, several of the simple shapes, such as boxes, cones, tubes (including their segmented forms), have been successfully ported to Vc code.

The GEANT-V project was thus able to show several improvements through their Vc reimplementations of existing geometry algorithms. Figures 12.4 & 12.5 show the results from vectorization of the geometry codes. It is clearly visible how Vc has enabled the GEANT-V project to achieve consistent efficiency improvements while expressing their parallel processing in a portable way.

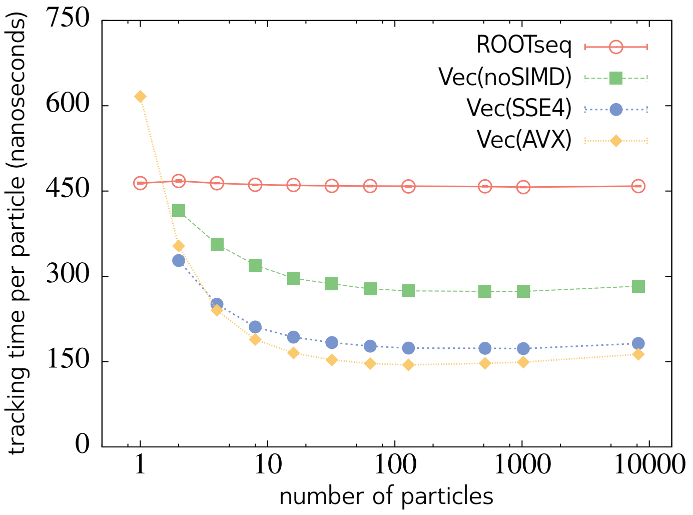


Figure 12.5: Results from the benchmark comparing the scalar algorithm (ROOT seq) with a vector-oriented algorithm using various degrees of usage of SIMD instructions [VEC(noSIMD), VEC(SSE4) and VEC(AVX)]. Comparison of the runtime per particle showing a speedup factor of roughly 3 comparing the original version to the AVX code.
[reproduced from 84]

12.4 ROOT

The ROOT library [8] is a de facto standard building block for the software of high energy physics experiments and many other physics projects. *About | ROOT* [1] summarizes the functionality: “The ROOT system provides a set of OO frameworks with all the functionality needed to handle and analyze large amounts of data in a very efficient way.” It is used by thousands of scientists world-wide and dozens of current experiments (e.g. ALICE, ATLAS, BaBar, CMS, COMPASS, LHCb, PHENIX, PHOBOS, STAR, …) use it for their software.

Since the 5.34/18 release of ROOT, the Vc library is included as a module. This enables Vc to reach a large audience. At the same time the inclusion of Vc in ROOT gives physics experiments an easier adoption strategy, since the ROOT library is already an accepted prerequisite.

12.5 CONCLUSION

On the basis of the ALICE usage I have been able to show how API improvements and extensions since previous Vc releases improve the applicability and flexibility of Vc. The ALICE example also has shown how there are often many possible approaches to vectorizing a given algorithm, each with different scaling behavior. Finally the ALICE example (as well as the k-d tree example in Section 11.3) has shown that Amdahl's law is relevant when considering vectorization opportunities.

This chapter has shown that the Vc library is in production use in large software projects and international communities. The ROOT, Geant-V, STAR, and CBM discussions have shown that Vc has become an important building block in software development for high performance computing in high energy physics experiments. This is an important achievement since it proves the portability and compatibility of the library.

Part IV

Conclusion

13 CONCLUSION

Chapter 1 has made the case that software developers and designers have to consider SIMD. Developers need to understand what SIMD can do and how it can be used in order to design data structures and algorithms which fully utilize modern CPUs. The presented vector and mask types (Part II) give software developers a flexible, portable, and efficient solution to explicit data-parallel programming. At the same time, through their API, these types guide the developer to a better understanding of efficient data-parallel execution and software design.

Chapter 2 has discussed the main alternative approach of loop transformations. I have shown the limitations, language restrictions, and semantic subtleties that are part of this approach. The data-parallel vector and mask types as presented in Part II overcome these limitations since they work inside the traditional C++ type system and follow the well-understood serial semantics of the *sequenced before* rule [48, §1.9 p13]. Chapter 4 has given a thorough discussion of the API for the vector type, which allows data-parallel execution for arithmetic types. The user does not have to reason about concurrent execution of serially stated algorithms but can reason about serially executing operations of which each processes data in parallel. Vc's `Vector<T>` type thus enables a data-parallel programming model which is easy to reason about without abandoning functionality.

In Chapter 5 I have then defined a type for vectors of booleans and have shown how these mask types can be used to develop data-dependent algorithms via write-masking. Therefore, in contrast to loop-vectorization or extended `if` statements in array notation, no *implicit* write-masking and branch flattening is necessary. The Vc interface consequently makes bad algorithm or data structure design visible. It helps users understand the cost of conditionals and facilitates a search for optimizations.

The remaining chapters of Part II have presented some of the higher abstractions that can be built upon the vector and mask types. These abstractions make data-parallel programming easier and more efficient.

The relevance of my research is evident in the interest of the concurrency study group of the C++ committee. Currently, the main obstacle to the study group is the implication for ABI compatibility. Therefore, Chapter 8 has discussed the challenge implementors of C++ are facing with regard to binary compatibility across microarchitectures. Despite this question there is consent in the study group to investigate the vector type programming model for use in standard C++. At the same time there is more research to be done in abstracting the vector width differences (i.e. enable programmers to write code without using the $\mathcal{W}_T$ constant). One of the abstraction ideas has been presented in Chapter 9, which has shown that the STL algorithms can be a good interface for vectorization and that vector types resolve the semantic issues of the current parallel algorithms proposal.

Chapter 10 has presented a solution for automatic vectorization of data structures and the template functions that work with these types. The `simdize<T>` expression enables the implementation of vectorized algorithms with a generic interface and supports developers to create data structures that deliver the best compromise between data locality and efficient vector loads and stores (or *AoS* vs. *SoA*).

Part III has discussed example applications of the Vc library. It has shown that vectorization of search data structures is possible with vector types, allowing explicit expression of data-parallel storage and execution. This is a major improvement over loop-vectorization, which only solves the programming language shortcoming of expressing data-parallel execution. I have also shown that the Vc library is in production use in large software projects and international communities. Vc has become an important building block in software development for high performance computing in high energy physics experiments. This is an important achievement since it proves the portability and compatibility of the library.

Part V

Appendix

A

WORKING-SET BENCHMARK

```
using Vc::float_v;

// This example shows how an arbitrary problem scales depending on working-set
// size and FLOPs per load/store. Understanding this can help to create better
// implementations.

// The Runner is a method to generate the different scenarios with all parameters
// to the Work available as constant expressions. The idea is to have the compiler
// able to optimize as much as possible so that the actual workload alone is
// benchmarked.

// The Runner recursively calls operator() on the Work template class with varying
// arguments for N and FLOPs.
template <template <std::size_t N, std::size_t M, int, int> class Work,
          std::size_t N = 256, std::size_t M = 4, int FLOPs = 2>
struct Runner {
  static void run() {
    Work<N, M, (N > 4096 ? 1 : 4096 / N), FLOPs>()();
    Runner<Work, N, M, int(FLOPs * 1.5)>::run();
  }
};
template <template <std::size_t N, std::size_t M, int, int> class Work,
          std::size_t N, std::size_t M>
struct Runner<Work, N, M, 211> {
  static void run() { Runner<Work, N * 2, M>::run(); }
};
template <template <std::size_t N, std::size_t M, int, int> class Work,
          std::size_t M, int FLOPs>
struct Runner<Work, 256 * 1024 * 1024, M, FLOPs> {
  static void run() {}
};

// The Flops helper struct generates code that executes FLOPs many floating-point
// SIMD instructions (add, sub, and mul)
template <int FLOPs> struct Flops {
  inline float_v operator()(float_v a, float_v b, float_v c) {
    typedef Flops<(FLOPs - 5) / 2> F1;
    typedef Flops<(FLOPs - 4) / 2> F2;
    return F1()(a + b, a * b, c) + F2()(a * c, b + c, a);
  }
};
```

```
template <> inline float_v Flops<2>::operator()(float_v a, float_v b, float_v c) {
  return a * b + c;
}
template <> inline float_v Flops<3>::operator()(float_v a, float_v b, float_v c) {
  return a * b + (c - a);
}
template <> inline float_v Flops<4>::operator()(float_v a, float_v b, float_v c) {
  return (a * b + c) + a * c;
}
template <> inline float_v Flops<5>::operator()(float_v a, float_v b, float_v c) {
  return a * b + (a + c) + a * c;
}
template <> inline float_v Flops<6>::operator()(float_v a, float_v b, float_v c) {
  return (a * b + (a + c)) + (a * c - b);
}
template <> inline float_v Flops<7>::operator()(float_v a, float_v b, float_v c) {
  return (a * b + (a + c)) + (a * c - (b + c));
}
template <> inline float_v Flops<8>::operator()(float_v a, float_v b, float_v c) {
  return (a * b + (a + c) + b) + (a * c - (b + c));
}

// This is the benchmark code. It is called from Runner and uses Flops to do the
// work.
template <std::size_t _N, std::size_t M, int Repetitions, int FLOPs>
struct ScaleWorkingSetSize {
  void operator()() {
    constexpr std::size_t N = _N / sizeof(float_v) + 3 * 16 / float_v::Size;
    typedef std::array<std::array<float_v, N>, M> Cont;
    auto data = Vc::make_unique<Cont, Vc::AlignOnPage>();
    for (auto &arr : *data) {
      for (auto &value : arr) {
        value = float_v::Random();
      }
    }

    TimeStampCounter tsc;
    double throughput = 0.;
    for (std::size_t i = 0; i < 2 + 512 / N; ++i) {
      tsc.start();
      // ------------- start of the benchmarked code ---------------
      for (int repetitions = 0; repetitions < Repetitions; ++repetitions) {
        for (std::size_t m = 0; m < M; ++m) {
          for (std::size_t n = 0; n < N; ++n) {
            (*data)[m][n] =
                Flops<FLOPs>()((*data)[(m + 1) % M][n], (*data)[(m + 2) % M][n],
                               (*data)[(m + 3) % M][n]);
          }
        }
      }
      // -------------- end of the benchmarked code ----------------
      tsc.stop();

      throughput =
          std::max(throughput, (Repetitions * M * N * float_v::Size * FLOPs) /
                                    static_cast<double>(tsc.cycles()));
    }

    const long bytes = N * M * sizeof(float_v);
    printf("%10lu Byte | %4.2f FLOP/Byte | %4.1f FLOP/cycle\n", bytes,
           static_cast<double>(float_v::Size * FLOPs) / (4 * sizeof(float_v)),
           throughput);
  }
};

int main() {
  ScaleWorkingSetSize<256, 4, 10, 2>()();  // dry-run before the real benchmark
  printf("%10s | %4s | %4s\n", "Working-Set Size", "FLOPs per Byte",
         "Throughput (FLOPs/Cycle)");
  Runner<ScaleWorkingSetSize>::run();
  return 0;
}
```

Listing A.1: The benchmark code for Figure 1.4.

B
LINEAR SEARCH BENCHMARK

```
int main() {
  std::cout << std::setw(15) << "N" << std::setw(15) << "std"
            << std::setw(15) << "stddev" << std::setw(15) << "Vc"
            << std::setw(15) << "stddev" << std::setw(15) << "speedup"
            << std::setw(15) << "stddev" << '\n';

  // create data
  std::vector<float, Vc::Allocator<float>> data;
  constexpr std::size_t NMax = 1024 * 128 * float_v::size();
  data.reserve(NMax);
  std::default_random_engine rne;
  std::uniform_real_distribution<float> uniform_dist(-1000.f, 1000.f);
  for (auto n = data.capacity(); n > 0; --n) {
    data.push_back(uniform_dist(rne));
  }

  for (std::size_t N = float_v::size() * 2; N <= NMax; N *= 2) {
    const std::size_t Repetitions = 100 + 1024 * 32 / N;

    // create search values
    std::vector<float> search_values;
    search_values.reserve(10000);
    for (auto n = search_values.capacity(); n > 0; --n) {
      search_values.push_back(
          data[std::uniform_int_distribution<std::size_t>(0, N - 1)(rne)]);
    }

    enum { std, vec };

    std::vector<decltype(data.begin())> iterators[2];
    iterators[std].resize(search_values.size());
    iterators[vec].resize(search_values.size());
    TimeStampCounter tsc;
    std::vector<decltype(tsc.cycles())> cycles[2];
    cycles[std].resize(Repetitions);
    cycles[vec].resize(Repetitions);

    double mean[2] = {};
    double stddev[2] = {};
    do {
      // search (std)
      for (auto n = Repetitions; n; --n) {
        tsc.start();
        for (std::size_t i = 0; i < search_values.size(); ++i) {
          iterators[std][i] =
              std::find(data.begin(), data.begin() + N, search_values[i]);
        }
        tsc.stop();
        cycles[std][Repetitions - n] = tsc.cycles();
      }

      // search (vec)
      for (auto n = Repetitions; n; --n) {
        tsc.start();
        for (std::size_t i = 0; i < search_values.size(); ++i) {
          iterators[vec][i] =
              Vc::find(data.begin(), data.begin() + N, search_values[i]);
        }
        tsc.stop();
        cycles[vec][Repetitions - n] = tsc.cycles();
      }

      // test that the results are equal
      for (std::size_t i = 0; i < iterators[vec].size(); ++i) {
        assert(iterators[std][i] == iterators[vec][i]);
      }

      // output results
      std::cout << std::setw(15) << N;
      for (int i : {std, vec}) {
        mean[i] = 0;
        stddev[i] = 0;
        std::sort(cycles[i].begin(), cycles[i].end());
        for (double x : cycles[i]) {
          mean[i] += x;
          stddev[i] += x * x;
        }
        mean[i] /= cycles[i].size();
        stddev[i] /= cycles[i].size();
        stddev[i] = std::sqrt(stddev[i] - mean[i] * mean[i]);
      }
      std::cout << std::setw(15) << mean[std] / search_values.size();
      std::cout << std::setw(15) << stddev[std] / search_values.size();
      std::cout << std::setw(15) << mean[vec] / search_values.size();
      std::cout << std::setw(15) << stddev[vec] / search_values.size();
      std::cout << std::setw(15) << std::setprecision(4) << mean[std] / mean[vec];
      std::cout << std::setw(15)
                << mean[std] / mean[vec] *
                       std::sqrt(
                           stddev[std] * stddev[std] / (mean[std] * mean[std]) +
                           stddev[vec] * stddev[vec] / (mean[vec] * mean[vec]));
      std::cout << std::endl;
    }  // if the error is large the system was busy doing something else and we
       // better repeat the experiment
    while (stddev[std] * 20 > mean[std] || stddev[vec] * 20 > mean[vec]);
  }
  return 0;
}
```

Listing B.1: The benchmark code for Figure 11.1.

C

NEAREST NEIGHBOR BENCHMARK

```
constexpr int NumberOfSearches = 10000;
constexpr int FirstSetSize = 64;
constexpr int MaxSetSize = 32 * 1024 * 1024;
constexpr int Repetitions = 100;
constexpr auto StepMultiplier = 2;
constexpr bool Optimize = true;
constexpr int Dimensions = 3;
using PointEntry = float;
constexpr PointEntry loCoord = -100;
constexpr PointEntry hiCoord = +100;

enum { KdS, KdV, LnS, LnV, NBenchmarks };

class Runner {
  const int SetSize;
  TimeStampCounter tsc;
  double mean[NBenchmarks] = {};
  double stddev[NBenchmarks] = {};

public:
  Runner(int S) : SetSize(S) {}
  void recordTsc(int Test, double norm) {
    const double x = tsc.cycles() / norm;
    mean[Test] += x;
    stddev[Test] += x * x;
  }
  void printRatio(int i, int j) {
    if (i >= 0 && j >= 0) {
      const auto ratio = mean[i] / mean[j];
      std::cout << std::setprecision(3) << std::setw(9) << ratio;
      std::cout << std::setprecision(3) << std::setw(9)
                << ratio * std::sqrt(stddev[i] * stddev[i] / (mean[i] * mean[i]) +
                                     stddev[j] * stddev[j] / (mean[j] * mean[j]));
    }
  }
  template <typename C, typename F>
  void benchmarkInsert(const int Test, C &&clear, F &&fun, double err = 5) {
    do {
      mean[Test] = 0;
      stddev[Test] = 0;
      for (auto rep = Repetitions; rep; --rep) {
        clear();
        tsc.start();
        fun();
        tsc.stop();
        recordTsc(Test, SetSize);
      }
      mean[Test] /= Repetitions;
      stddev[Test] /= Repetitions;
      stddev[Test] = std::sqrt(stddev[Test] - mean[Test] * mean[Test]);
    } while (stddev[Test] * err > mean[Test]);
    std::cout << std::setw(9) << std::setprecision(3) << mean[Test];
    std::cout << std::setw(9) << std::setprecision(3) << stddev[Test];
    std::cout << std::flush;
  }
  template <typename F>
  void benchmarkSearch(const int Test, F &&fun, double err = 20) {
    do {
      mean[Test] = 0;
      stddev[Test] = 0;
      fun();  // one cache warm-up run:
      for (auto rep = Repetitions; rep; --rep) {
        tsc.start();
        fun();
        tsc.stop();
        recordTsc(Test, NumberOfSearches);
      }
      mean[Test] /= Repetitions;
      stddev[Test] /= Repetitions;
      stddev[Test] = std::sqrt(stddev[Test] - mean[Test] * mean[Test]);
    } while (stddev[Test] * err > mean[Test]);
    std::cout << std::setw(9) << std::setprecision(3) << mean[Test];
    std::cout << std::setw(9) << std::setprecision(3) << stddev[Test];
    std::cout << std::flush;
  }
};

template <typename Container, typename F, std::size_t... Indexes>
void emplace_helper(Container &c, F &&f, Vc::index_sequence<Indexes...>) {
  c.emplace_back(f(Indexes)...);
}

int main() {
  using Point = ::Point<PointEntry, Dimensions>;

  // output header
  using std::cout;
  using std::setw;
  using std::setprecision;
  cout << "NumberOfSearches: " << NumberOfSearches << '\n';
  cout << "T: " << typeid(Point).name() << '\n';
  cout << "Volume: ";
  for (auto i = Dimensions; i; --i) {
    cout << hiCoord - loCoord << ((i > 1) ? "  " : "\n");
  }
  cout << setw(8) << 'N' << setw(18) << "KdTree" << setw(18) << "KdTreeV"
```

```
     << setw(18) << "Linear" << setw(18) << "LinearV" << setw(18) << "Kd/KdV"
     << setw(18) << "Lin/LinV";
{
  constexpr auto Width = Optimize ? 6 : 3;
  cout << setw(Width) << "DS" << setw(Width) << "DV";
}
cout << setw(18) << "KdTree" << setw(18) << "KdTreeV" << setw(18) << "Linear"
     << setw(18) << "LinearV" << setw(18) << "Kd/KdV" << setw(18) << "Lin/LinV"
     << std::endl;

// random haystack values and search points
std::default_random_engine randomEngine(1);
typename std::conditional<std::is_floating_point<PointEntry>::value,
                          std::uniform_real_distribution<PointEntry>,
                          std::uniform_int_distribution<PointEntry>>::type
    uniform(loCoord, hiCoord);
std::vector<Point> randomPoints;
randomPoints.reserve(MaxSetSize);
typedef Vc::make_index_sequence<Dimensions> InitSequence;
for (int i = 0; i < MaxSetSize; ++i) {
  emplace_helper(randomPoints, [&uniform, &randomEngine](int) {
                                 return uniform(randomEngine);
                               },
                 InitSequence());
}
std::vector<Point> searchPoints;
searchPoints.reserve(NumberOfSearches);
for (int i = 0; i < NumberOfSearches; ++i) {
  emplace_helper(searchPoints, [&uniform, &randomEngine](int) {
                                 return uniform(randomEngine);
                               },
                 InitSequence());
}

for (int SetSize = FirstSetSize; SetSize <= MaxSetSize;
     SetSize *= StepMultiplier) {
  cout << setw(8) << SetSize << std::flush;
  Runner testRunner(SetSize);

  KdTree<Point> pointsTree;
  testRunner.benchmarkInsert(KdS, [&]() { pointsTree.clear(); },
                             [&]() {
                               for (int i = 0; i < SetSize; ++i) {
                                 pointsTree.insert(randomPoints[i]);
                               }
                             });

  KdTreeV<Point> pointsTreeV;
  testRunner.benchmarkInsert(KdV, [&]() { pointsTreeV.clear(); },
                             [&]() {
                               for (int i = 0; i < SetSize; ++i) {
                                 pointsTreeV.insert(randomPoints[i]);
                               }
                             });
  LinearNeighborSearch<Point> linearSearch(SetSize);
  testRunner.benchmarkInsert(LnS, [&]() { linearSearch.clear(); },
                             [&]() {
                               for (int i = 0; i < SetSize; ++i) {
                                 linearSearch.insert(randomPoints[i]);
                               }
                             });

  LinearNeighborSearchV<Point> linearSearchV(SetSize);
  testRunner.benchmarkInsert(LnV, [&] { linearSearchV.clear(); },
                             [&] {
                               for (int i = 0; i < SetSize; ++i) {
                                 linearSearchV.insert(randomPoints[i]);
                               }
                             });

  testRunner.printRatio(KdS, KdV);
  testRunner.printRatio(LnS, LnV);

  cout << setw(3) << pointsTree.depth();
  if (Optimize) {
    pointsTree.optimize();
    cout << setw(3) << pointsTree.depth();
  }
  cout << setw(3) << pointsTreeV.depth();
  if (Optimize) {
    pointsTreeV.optimize();
    cout << setw(3) << pointsTreeV.depth();
  }
  cout << std::flush;

  testRunner.benchmarkSearch(KdS, [&] {
    for (int i = 0; i < NumberOfSearches; ++i) {
      const auto &p = searchPoints[i];
      const auto &p2 = pointsTree.findNearest(p);
      asm("" ::"m"(p2));
    }
  }, 15);
  testRunner.benchmarkSearch(KdV, [&] {
    for (int i = 0; i < NumberOfSearches; ++i) {
      const auto &p = searchPoints[i];
      const auto &p2 = pointsTreeV.findNearest(p);
      asm("" ::"m"(p2));
    }
  });
  testRunner.benchmarkSearch(LnS, [&]() {
    for (int i = 0; i < NumberOfSearches; ++i) {
      const auto &p = searchPoints[i];
      const auto &p2 = linearSearch.findNearest(p);
      asm("" ::"m"(p2));
    }
  });
  testRunner.benchmarkSearch(LnV, [&]() {
    for (int i = 0; i < NumberOfSearches; ++i) {
      const auto &p = searchPoints[i];
      const auto &p2 = linearSearchV.findNearest(p);
      asm("" ::"m"(p2));
    }
  });

  testRunner.printRatio(KdS, KdV);
  testRunner.printRatio(LnS, LnV);
  cout << std::endl;
}
return 0;
}
```

Listing C.1: The benchmark code for Figures 11.3, 11.6, 11.7, 11.8, and 11.9.

D

BENCHMARKING

Listing D.1 shows the script I used before executing any of the benchmarks shown in this document. The script tests for the existence of a Turbo Modus of the CPU, and disables it if present. On newer Linux kernels with an Intel CPU the Turbo Modus is controlled via the *Intel P-state driver* [7]. In this case a simple boolean switch is exposed by the driver to disable the Turbo Modus. Otherwise, the Turbo Modus is identified in the `cpufreq` kernel module as a frequency which is 1 MHz higher than the nominal frequency of the CPU. Additionally, the scaling governor is set to performance mode, to alleviate the need for warm-up phases in the benchmark.

```sh
#!/bin/sh
cd /sys/devices/system/cpu
no_turbo=intel_pstate/no_turbo
if test -f $no_turbo; then
  echo 1 > $no_turbo
else
  freq=$(cut -d" " -f1,2 cpu0/cpufreq/scaling_available_frequencies)
  freq1=${freq% *}
  freq2=${freq#* }
  test $(($freq2+1000)) -eq $freq1 && \
    echo $freq2 | tee cpu[0-9]*/cpufreq/scaling_max_freq >/dev/null
  echo performance | tee cpu[0-9]*/cpufreq/scaling_governor >/dev/null
fi
```

Listing D.1: Shell script enabling reproducible benchmarks on current x86 CPUs.

E

C++ IMPLEMENTATIONS OF THE K-D TREE ALGORITHMS

E.1 *INSERT* ALGORITHM

```
template <typename T, std::size_t Dimensions>
void KdTree<T, Dimensions>::insert(const T &x) {
  if (root) {
    root->insert(x);
  } else {
    root = make_unique<Node<0>>(x);
  }
}

template <typename T, std::size_t Dimensions>
template <std::size_t SplittingPlane>
void KdTree<T, Dimensions>::Node<SplittingPlane>::insert(const T &x) {
  auto &child = (get_kdtree_value<SplittingPlane>(x) <
                 get_kdtree_value<SplittingPlane>(data))
                    ? child[0]
                    : child[1];
  if (child) {
    child->insert(x);
  } else {
    child = make_unique<Node<ChildSplittingPlane>>(x);
  }
}
```

Listing E.1: The C++ implementation of the scalar k-d tree insert algorithm.

```
template <typename T, std::size_t Dimensions>
void KdTreeV<T, Dimensions>::insert(const T &x) {
  if (root) {
    root->insert(x);
  } else {
    root = make_unique<Node<0>>(x);
  }
}

template <typename T, std::size_t Dimensions>
template <std::size_t SplittingPlane>
void KdTreeV<T, Dimensions>::Node<SplittingPlane>::insert(const T &x) {
  using namespace std;
  if (entries < V::Size) {
    const auto less = get_kdtree_value<SplittingPlane>(V(x)) <
                      get_kdtree_value<SplittingPlane>(v());
    if (any_of(less)) {
      where(less.firstOne() <= VEntry::IndexesFromZero()) | v() =
          iif(less.firstOne() == VEntry::IndexesFromZero(), xv,
              simdize_shifted(v(), -1));
    } else {
      // if none_of(x < this) then x will be the largest value in this node
      simdize_assign(v(), entries, x);
    }
    ++entries;
  } else {
    const auto less = get_kdtree_value<SplittingPlane>(v()) <
                      get_kdtree_value<SplittingPlane>(V(x));
    if (all_of(less)) {  // go to the lo child
      childInsert(child[1], x);
    } else if (none_of(less)) {  // go to the hi child
      childInsert(child[0], x);
    } else {
      int i = less.count();
      if (i > V::size() / 2) {
        const T xx = simdize_extract(v(), V::size() - 1);
        where(i <= VEntry::IndexesFromZero()) | v() =
            iif(i == VEntry::IndexesFromZero(), xv, simdize_shifted(v(), -1));
        childInsert(m_child[1], xx);
      } else {
        const T xx = simdize_extract(v(), 0);
        where(i >= VEntry::IndexesFromZero()) | v() =
            iif(i == VEntry::IndexesFromZero(), xv, simdize_shifted(v(), 1));
        childInsert(m_child[0], xx);
      }
    }
  }
}

template <typename T, std::size_t Dimensions>
template <std::size_t SplittingPlane>
void KdTreeV<T, Dimensions>::Node<SplittingPlane>::childInsert(ChildPtr &child,
                                                               const T new_x) {
  if (child) {
    child->insert(new_x);
  } else {
    child = make_unique<Node<ChildSplittingPlane>>(new_x);
  }
}
```

Listing E.2: The C++ & Vc implementation of the vectorized k-d tree insert algorithm. The broadcast of `new_x` to all entries in the vector is implemented in the **`Node`** constructor.

E.2 *FINDNEAREST* ALGORITHM

```
template <typename T, std::size_t Dimensions>
T KdTree<T, Dimensions>::findNearest(T x) const {
  if (!root) {
    throw std::runtime_error(
        "No values in the KdTree, which is required for findNearest.");
  }
  CandidateType nearest =
      std::make_pair(T(), std::numeric_limits<DistanceType>::max());
  m_root->findNearest(x, nearest);
  return nearest.first;
}

template <typename T, std::size_t Dimensions>
template <std::size_t SplittingPlane>
void KdTree<T, Dimensions>::Node<SplittingPlane>::findNearest(
    const T &x, CandidateType &candidate) const {
  // terminal node: return best distance from this node
  if (child[0] == child[1]) {
    const auto distance = get_kdtree_distance(x, data);
    if (distance < candidate.second) {
      candidate = std::make_pair(data, distance);
    }
    return;
  }

  const auto dx = get_kdtree_1dim_distance<SplittingPlane>(x, data);

  const std::size_t index = (get_kdtree_value<SplittingPlane>(x) <
                             get_kdtree_value<SplittingPlane>(data))
                                ? 0
                                : 1;
  if (child[index]) {
    // if we have a child node on the side where the search point would get
    // stored, that node is an obvious candidate
    child[index]->findNearest(x, candidate);
  }
  if (dx < candidate.second) {
    const auto distance = get_kdtree_distance(x, data);
    if (distance < candidate.second) {
      candidate = std::make_pair(data, distance);
      if (dx >= candidate.second) {
        return;
      }
    }
    if (child[index ^ 1]) {
      // if we have a child on the "wrong" side it could still be/find the nearest
      // neighbor, but only if the shortest distance of the search point x to the
      // splitting plane is less than the distance to the current candidate.
      child[index ^ 1]->findNearest(x, candidate);
    }
  }
}
```

Listing E.3: The C++ implementation of the scalar k-d tree nearest neighbor search algorithm.

```cpp
struct CandidateType {
  V nearest;
  DistanceTypeV distance =
      DistanceTypeV(std::numeric_limits<DistanceType>::max());
  T min() const {
    return simdize_extract(nearest, (distance == distance.min()).firstOne());
  }
};

template <typename T, std::size_t Dimensions>
T KdTreeV<T, Dimensions>::findNearest(T x) const {
  if (!root) {
    throw std::runtime_error(
        "No values in the KdTree, which is required for findNearest.");
  }
  CandidateType nearest;
  root->findNearest(V(x), nearest);
  return nearest.min();
}

template <typename T, std::size_t Dimensions>
template <std::size_t SplittingPlane>
std::pair<T, DistanceType> KdTreeV<T, Dimensions>::Node<
    SplittingPlane>::findNearest(const V &x, CandidateType &candidate) const {
  // guess near and far child nodes
  auto nearChild = child[0].get();
  auto farChild = child[1].get();

  if (nearChild == farChild) {  // iff both are nullptr
    // leaf node. 50% of nodes in a balanced tree are leaf nodes.
    const auto distance = get_kdtree_distance(x, v());
    if (Dimensions < 4 || any_of(distance < candidate.distance)) {
      where(distance < candidate.distance) | candidate.nearest = v();
      candidate.distance = min(candidate.distance, distance);
    }
    return;
  } else if (nearChild && farChild) {
    // The other 50% of nodes in a balanced tree have two children.
    const auto left = get_kdtree_value<SplittingPlane>(v())[0];
    const auto right = get_kdtree_value<SplittingPlane>(v())[V::size() - 1];
    const auto xx = get_kdtree_value<SplittingPlane>(x)[0];

    DistanceTypeV dxMax = get_kdtree_1dim_distance(xx, right);
    DistanceTypeV dxMin = get_kdtree_1dim_distance(xx, left);

    if (xx >= right) {
      std::swap(dxMax, dxMin);
      std::swap(nearChild, farChild);
    } else if (xx > left) {          // xx is between left & right
      if (left + right < xx * 2) {  // discriminate on center
```

```
        std::swap(dxMax, dxMin);
        std::swap(nearChild, farChild);
      }
      const auto distance = get_kdtree_distance(x, v());
      if (Dimensions < 4 || any_of(distance < candidate.distance)) {
        where(distance < candidate.distance) | candidate.nearest = v();
        candidate.distance = min(candidate.distance, distance);
      }
      if (all_of(dxMin < candidate.distance)) {
        nearChild->findNearest(x, candidate);
      }
      if (all_of(dxMax < candidate.distance)) {
        farChild->findNearest(x, candidate);
      }
      return;
    }
    nearChild->findNearest(x, candidate);
    if (all_of(dxMin < candidate.distance)) {
      const auto distance = get_kdtree_distance(x, v());
      if (Dimensions < 4 || any_of(distance < candidate.distance)) {
        where(distance < candidate.distance) | candidate.nearest = v();
        candidate.distance = min(candidate.distance, distance);
      }
      if (all_of(dxMax < candidate.distance)) {
        farChild->findNearest(x, candidate);
      }
    }
  } else if (child[0]) {
    const auto left = get_kdtree_value<SplittingPlane>(v())[0];
    const auto xx = get_kdtree_value<SplittingPlane>(x)[0];
    const auto dx = get_kdtree_1dim_distance(xx, left);
    if (xx < left) {
      child[0]->findNearest(x, candidate);
      if (all_of(dx < candidate.distance)) {
        const auto distance = get_kdtree_distance(x, v());
        if (Dimensions < 4 || any_of(distance < candidate.distance)) {
          where(distance < candidate.distance) | candidate.nearest = v();
          candidate.distance = min(candidate.distance, distance);
        }
      }
    } else {
      const auto distance = get_kdtree_distance(x, v());
      if (Dimensions < 4 || any_of(distance < candidate.distance)) {
        where(distance < candidate.distance) | candidate.nearest = v();
        candidate.distance = min(candidate.distance, distance);
      }
      if (all_of(dx < candidate.distance)) {
        child[0]->findNearest(x, candidate);
      }
    }
  } else {  // only child[1]
    const auto right = get_kdtree_value<SplittingPlane>(v())[V::size() - 1];
    const auto xx = get_kdtree_value<SplittingPlane>(x)[0];
    const auto dx = get_kdtree_1dim_distance(xx, right);
    if (xx > right) {
      child[1]->findNearest(x, candidate);
      if (all_of(dx < candidate.distance)) {
        const auto distance = get_kdtree_distance(x, v());
        if (Dimensions < 4 || any_of(distance < candidate.distance)) {
          where(distance < candidate.distance) | candidate.nearest = v();
          candidate.distance = min(candidate.distance, distance);
        }
      }
    } else {
      const auto distance = get_kdtree_distance(x, v());
      if (Dimensions < 4 || any_of(distance < candidate.distance)) {
        where(distance < candidate.distance) | candidate.nearest = v();
        candidate.distance = min(candidate.distance, distance);
      }
      if (all_of(dx < candidate.distance)) {
        child[1]->findNearest(x, candidate);
      }
    }
  }
}
```

Listing E.4: The C++ & Vc implementation of the vectorized k-d tree nearest neighbor search algorithm.

```
template <typename T, std::size_t Dimensions>
simdize<T> KdTree<T, Dimensions>::findNearest(const simdize<T> &x) const {
  if (!m_root) {
    throw std::runtime_error(
        "No values in the KdTree, which is required for findNearest.");
  }

  CandidateTypeV nearest = std::make_pair(
      simdize<T>(), DistanceTypeV(std::numeric_limits<DistanceType>::max()));
  m_root->findNearest(x, nearest);
  return nearest.first;
}

template <typename T, std::size_t Dimensions>
template <std::size_t SplittingPlane>
void KdTree<T, Dimensions>::Node<SplittingPlane>::findNearest(
    const simdize<T> &x, CandidateTypeV &candidate) const {
  using V = simdize<T>;
  V data_v(m_data);
  if (m_child[0] == m_child[1]) {
    const auto distance = get_kdtree_distance(x, data_v);
    where(distance < candidate.second) | candidate.first = data_v;
    candidate.second = min(distance, candidate.second);
    return;
  }

  const auto dx = get_kdtree_1dim_distance<SplittingPlane>(x, data_v);
  const bool indexDisagrees = (get_kdtree_value<SplittingPlane>(x) <
                               get_kdtree_value<SplittingPlane>(data_v)).count() %
                                  x.size() !=
                              0;
  const std::size_t index =
      (get_kdtree_value<SplittingPlane>(x) <
       get_kdtree_value<SplittingPlane>(data_v)).count() > x.size() / 2
          ? 0
          : 1;
  if (m_child[index]) {
    m_child[index]->findNearest(x, candidate);
  }
  if (indexDisagrees) {
    if (m_child[index ^ 1]) {
      m_child[index ^ 1]->findNearest(x, candidate);
    }
    const auto distance = get_kdtree_distance(x, data_v);
    where(distance < candidate.second) | candidate.first = data_v;
    candidate.second = min(distance, candidate.second);
  } else if (any_of(dx < candidate.second)) {
    const auto distance = get_kdtree_distance(x, data_v);
    where(distance < candidate.second) | candidate.first = data_v;
    candidate.second = min(distance, candidate.second);
    if (all_of(dx >= candidate.second)) {
      return;
    }
    if (m_child[index ^ 1]) {
      m_child[index ^ 1]->findNearest(x, candidate);
    }
  }
}
```

Listing E.5: The C++ & Vc implementation of vectorized nearest neighbor search in a scalar k-d tree.

F

ALICE SPLINE SOURCES

F.1 SPLINE IMPLEMENTATIONS

```
typedef std::array<float, 2> Point2;
typedef std::array<float, 3> Point3;
typedef simdize<Point2> Point2V;
typedef simdize<Point3> Point3V;

class Spline /* or Spline2 or Spline3*/ {
  const int fNA;        // # points A axis
  const int fNB;        // # points B axis
  const int fN;         // # points total
  const float fMinA;    // min A axis
  const float fMinB;    // min B axis
  const float fStepA;   // step between points A axis
  const float fStepB;   // step between points B axis
  const float fScaleA;  // scale A axis
  const float fScaleB;  // scale B axis

  Vc::vector<Vc::SimdArray<float, 4>> fXYZ;  // Spline
  // Spline2: Vc::vector<float> fXYZ;
  // Spline3: Vc::vector<std::array<float, 3>> fXYZ;
};

inline void Spline::Fill(int ind, float x, float y, float z) {
  fXYZ[ind][0] = x;
  fXYZ[ind][1] = y;
  fXYZ[ind][2] = z;
}

inline void Spline2::Fill(int ind, float x, float y, float z) {
  ind = ind / fNB + fNA * (ind % fNB);
  fXYZ[ind] = x;
  fXYZ[ind + fN] = y;
  fXYZ[ind + 2 * fN] = z;
}

inline void Spline3::Fill(int ind, float x, float y, float z) {
  ind = ind / fNB + fNA * (ind % fNB);
  fXYZ[ind][0] = x;
  fXYZ[ind][1] = y;
  fXYZ[ind][2] = z;
}
```

Listing F.1: The class layouts for the different spline implementations.

```
inline std::tuple<int, int, float, float> evaluatePosition(Point2 ab, Point2 min,
                                                            Point2 scale, int na,
                                                            int nb) {
  const float lA = (ab[0] - min[0]) * scale[0] - 1.f;
  const int iA = std::min(na - 4.f, std::max(lA, 0.f));
  const float lB = (ab[1] - min[1]) * scale[1] - 1.f;
  const int iB = std::min(nb - 4.f, std::max(lB, 0.f));
  const float da = lA - iA;
  const float db = lB - iB;
  return std::make_tuple(iA, iB, da, db);
}

using Vc::float_v;
typedef float_v::IndexType index_v;
inline std::tuple<index_v, index_v, float_v, float_v> evaluatePosition(
    Point2V ab, Point2 min, Point2 scale, int na, int nb) {
  const float_v lA = (ab[0] - min[0]) * scale[0] - 1.f;
  const auto iA = static_cast<index_v>(std::min(na - 4.f, std::max(lA, 0.f)));
  const float_v lB = (ab[1] - min[1]) * scale[1] - 1.f;
  const auto iB = static_cast<index_v>(std::min(nb - 4.f, std::max(lB, 0.f)));
  const float_v da = lA - Vc::simd_cast<float_v>(iA);
  const float_v db = lB - Vc::simd_cast<float_v>(iB);
  return std::make_tuple(iA, iB, da, db);
}
```

Listing F.2: The map index calculation common to the different spline implementations.

```
Point3 Spline::GetValue(Point2 ab) const {
  float da1, db1;
  int iA, iB;
  std::tie(iA, iB, da1, db1) =
      evaluatePosition(ab, {fMinA, fMinB}, {fScaleA, fScaleB}, fNA, fNB);
  int ind = iA * fNB + iB;
  typedef Vc::SimdArray<float, 4> float4;
  const float4 da = da1;
  const float4 db = db1;
  float4 v[4];
  const float4 *m = &fXYZ[0];
  for (int i = 0; i < 4; i++) {
    v[i] = GetSpline3(m[ind + 0], m[ind + 1], m[ind + 2], m[ind + 3], db);
    ind += fNB;
  }
  float4 res = GetSpline3(v[0], v[1], v[2], v[3], da);
  return {res[0], res[1], res[2]};
}
```

Listing F.3: FLOAT4 spline implementation.

```
Point3 Spline::GetValue16(Point2 ab) const {
  float da1, db1;
  int iA, iB;
  std::tie(iA, iB, da1, db1) =
      evaluatePosition(ab, {fMinA, fMinB}, {fScaleA, fScaleB}, fNA, fNB);
  typedef Vc::SimdArray<float, 4> float4;
  typedef Vc::SimdArray<float, 16> float16;
  const float4 da = da1;
  const float16 db = db1;
  const float4 *m0 = &fXYZ[iA * fNB + iB];
  const float4 *m1 = m0 + fNB;
  const float4 *m2 = m1 + fNB;
  const float4 *m3 = m2 + fNB;
  const float16 v0123 =
      GetSpline3(Vc::simd_cast<float16>(m0[0], m1[0], m2[0], m3[0]),
                 Vc::simd_cast<float16>(m0[1], m1[1], m2[1], m3[1]),
                 Vc::simd_cast<float16>(m0[2], m1[2], m2[2], m3[2]),
                 Vc::simd_cast<float16>(m0[3], m1[3], m2[3], m3[3]), db);
  const float4 res = GetSpline3(
      Vc::simd_cast<float4, 0>(v0123), Vc::simd_cast<float4, 1>(v0123),
      Vc::simd_cast<float4, 2>(v0123), Vc::simd_cast<float4, 3>(v0123), da);
  return {res[0], res[1], res[2]};
}
```

Listing F.4: FLOAT16 spline implementation.

```
inline Point3 Spline2::GetValue(Point2 ab) const {
  float da1, db1;
  int iA, iB;
  std::tie(iA, iB, da1, db1) =
      evaluatePosition(ab, {fMinA, fMinB}, {fScaleA, fScaleB}, fNA, fNB);
  typedef Vc::SimdArray<float, 4> float4;
  typedef Vc::SimdArray<float, 12> float12;
  const float4 da = da1;
  const float12 db = db1;
  const float *m0 = &fXYZ[iA + iB * fNA];
  const float *m1 = m0 + fNA;
  const float *m2 = m1 + fNA;
  const float *m3 = m2 + fNA;
  const float12 xyz = GetSpline3(
      Vc::simd_cast<float12>(float4(m0), float4(m0 + fN), float4(m0 + 2 * fN)),
      Vc::simd_cast<float12>(float4(m1), float4(m1 + fN), float4(m1 + 2 * fN)),
      Vc::simd_cast<float12>(float4(m2), float4(m2 + fN), float4(m2 + 2 * fN)),
      Vc::simd_cast<float12>(float4(m3), float4(m3 + fN), float4(m3 + 2 * fN)),
      db);
  float4 v[4];
  Vc::tie(v[0], v[1], v[2], v[3]) =
      Vc::transpose(Vc::simd_cast<float4, 0>(xyz), Vc::simd_cast<float4, 1>(xyz),
                    Vc::simd_cast<float4, 2>(xyz), float4::Zero());
  float4 res = GetSpline3(v[0], v[1], v[2], v[3], da);
  return {res[0], res[1], res[2]};
}
```

Listing F.5: FLOAT12 spline implementation.

```
inline Point3 Spline3::GetValue(Point2 ab) const {
  float da1, db1;
  int iA, iB;
  std::tie(iA, iB, da1, db1) =
      evaluatePosition(ab, {fMinA, fMinB}, {fScaleA, fScaleB}, fNA, fNB);
  typedef Vc::SimdArray<float, 4> float4;
  typedef Vc::SimdArray<float, 12> float12;
  const float4 da = da1;
  const float12 db = db1;
  const float *m0 = &fXYZ[iA + iB * fNA][0];
  const float *m1 = m0 + fNA * 3;
  const float *m2 = m1 + fNA * 3;
  const float *m3 = m2 + fNA * 3;
  const float12 xyz =
      GetSpline3(float12(m0), float12(m1), float12(m2), float12(m3), db);
  const float4 t0 = Vc::simd_cast<float4, 0>(xyz);
  const float4 t1 = Vc::simd_cast<float4, 1>(xyz);
  const float4 t2 = Vc::simd_cast<float4, 2>(xyz);
  const float4 res =
      GetSpline3(t0, t0.shifted(3, t1), t1.shifted(2, t2), t2.shifted(1), da);
  return {res[0], res[1], res[2]};
}
```

Listing F.6: FLOAT12 INTERLEAVED spline implementation.

```
Point3V Spline::GetValue(const Point2V &ab) const {
  index_v iA, iB;
  float_v da, db;
  std::tie(iA, iB, da, db) =
      evaluatePosition(ab, {fMinA, fMinB}, {fScaleA, fScaleB}, fNA, fNB);
  float_v vx[4];
  float_v vy[4];
  float_v vz[4];
  auto ind = iA * fNB + iB;
  const auto map = Vc::make_interleave_wrapper<float_v>(&fXYZ[0]);
  for (int i = 0; i < 4; i++) {
    float_v x[4], y[4], z[4];
    Vc::tie(x[0], y[0], z[0]) = map[ind];
    Vc::tie(x[1], y[1], z[1]) = map[ind + 1];
    Vc::tie(x[2], y[2], z[2]) = map[ind + 2];
    Vc::tie(x[3], y[3], z[3]) = map[ind + 3];
    vx[i] = GetSpline3<float_v>(x[0], x[1], x[2], x[3], db);
    vy[i] = GetSpline3<float_v>(y[0], y[1], y[2], y[3], db);
    vz[i] = GetSpline3<float_v>(z[0], z[1], z[2], z[3], db);
    ind += fNB;
  }
  Point3V XYZ;
  XYZ[0] = GetSpline3<float_v>(vx, da);
  XYZ[1] = GetSpline3<float_v>(vy, da);
  XYZ[2] = GetSpline3<float_v>(vz, da);
  return XYZ;
}
```

Listing F.7: HORIZONTAL 1 spline implementation.

```
inline Spline2::Point3V Spline2::GetValue(Point2V ab) const {
  index_v iA, iB;
  float_v da, db;
  std::tie(iA, iB, da, db) =
      evaluatePosition(ab, {fMinA, fMinB}, {fScaleA, fScaleB}, fNA, fNB);
  auto ind = iA + iB * fNA;
  Point3V xyz;
  {
    float_v x[4][4];
    Vc::tie(x[0][0], x[1][0], x[2][0], x[3][0]) = fXYZ[ind];
    Vc::tie(x[0][1], x[1][1], x[2][1], x[3][1]) = fXYZ[ind + fNA];
    Vc::tie(x[0][2], x[1][2], x[2][2], x[3][2]) = fXYZ[ind + 2 * fNA];
    Vc::tie(x[0][3], x[1][3], x[2][3], x[3][3]) = fXYZ[ind + 3 * fNA];
    xyz[0] = GetSpline3(GetSpline3(x[0], db), GetSpline3(x[1], db),
                        GetSpline3(x[2], db), GetSpline3(x[3], db), da);
  }
  ind += fN;
  {
    float_v y[4][4];
    Vc::tie(y[0][0], y[1][0], y[2][0], y[3][0]) = fXYZ[ind];
    Vc::tie(y[0][1], y[1][1], y[2][1], y[3][1]) = fXYZ[ind + fNA];
    Vc::tie(y[0][2], y[1][2], y[2][2], y[3][2]) = fXYZ[ind + 2 * fNA];
    Vc::tie(y[0][3], y[1][3], y[2][3], y[3][3]) = fXYZ[ind + 3 * fNA];
    xyz[1] = GetSpline3(GetSpline3(y[0], db), GetSpline3(y[1], db),
                        GetSpline3(y[2], db), GetSpline3(y[3], db), da);
  }
  ind += fN;
  {
    float_v z[4][4];
    Vc::tie(z[0][0], z[1][0], z[2][0], z[3][0]) = fXYZ[ind];
    Vc::tie(z[0][1], z[1][1], z[2][1], z[3][1]) = fXYZ[ind + fNA];
    Vc::tie(z[0][2], z[1][2], z[2][2], z[3][2]) = fXYZ[ind + 2 * fNA];
    Vc::tie(z[0][3], z[1][3], z[2][3], z[3][3]) = fXYZ[ind + 3 * fNA];
    xyz[2] = GetSpline3(GetSpline3(z[0], db), GetSpline3(z[1], db),
                        GetSpline3(z[2], db), GetSpline3(z[3], db), da);
  }
  return xyz;
}
```

Listing F.8: Horizontal 2 spline implementation.

```
Point3 Spline::GetValueAlice(Point2 ab) const {
  float lA = (ab[0] - fMinA) * fScaleA - 1.f;
  int iA = (int)lA;
  if (lA < 0)
    iA = 0;
  else if (iA > fNA - 4)
    iA = fNA - 4;
  float lB = (ab[1] - fMinB) * fScaleB - 1.f;
  int iB = (int)lB;
  if (lB < 0)
    iB = 0;
  else if (iB > fNB - 4)
    iB = fNB - 4;
  Point3 XYZ;
  if (Vc::float_v::Size == 4) {
    Vc::float_v da = lA - iA;
    Vc::float_v db = lB - iB;
    Vc::float_v v[4];
    int ind = iA * fNB + iB;
    const Vc::float_v *m = reinterpret_cast<const Vc::float_v *>(&fXYZ[0]);
    for (int i = 0; i < 4; i++) {
      v[i] = GetSpline3(m[ind + 0], m[ind + 1], m[ind + 2], m[ind + 3], db);
      ind += fNB;
    }
    Vc::float_v res = GetSpline3(v[0], v[1], v[2], v[3], da);
    XYZ[0] = res[0];
    XYZ[1] = res[1];
    XYZ[2] = res[2];
  } else {
    float da = lA - iA;
    float db = lB - iB;
    float vx[4];
    float vy[4];
    float vz[4];
    int ind = iA * fNB + iB;
    const float *m = reinterpret_cast<const float *>(&fXYZ[0]);
    for (int i = 0; i < 4; i++) {
      int ind4 = ind * 4;
      vx[i] = GetSpline3(m[ind4 + 0], m[ind4 + 4], m[ind4 + 8], m[ind4 + 12], db);
      vy[i] = GetSpline3(m[ind4 + 1], m[ind4 + 5], m[ind4 + 9], m[ind4 + 13], db);
      vz[i] = GetSpline3(m[ind4 + 2], m[ind4 + 6], m[ind4 +10], m[ind4 + 14], db);
      ind += fNB;
    }
    XYZ[0] = GetSpline3(vx, da);
    XYZ[1] = GetSpline3(vy, da);
    XYZ[2] = GetSpline3(vz, da);
  }
  return XYZ;
}
```

Listing F.9: Alice spline implementation.

```
Point3 Spline::GetValueScalar(Point2 ab) const {
  float da, db;
  int iA, iB;
  std::tie(iA, iB, da, db) =
      evaluatePosition(ab, {fMinA, fMinB}, {fScaleA, fScaleB}, fNA, fNB);
  int ind = iA * fNB + iB;
  float vx[4];
  float vy[4];
  float vz[4];
  for (int i = 0; i < 4; i++) {
    vx[i] = GetSpline3(fXYZ[ind][0], fXYZ[ind + 1][0], fXYZ[ind + 2][0],
                       fXYZ[ind + 3][0], db);
    vy[i] = GetSpline3(fXYZ[ind][1], fXYZ[ind + 1][1], fXYZ[ind + 2][1],
                       fXYZ[ind + 3][1], db);
    vz[i] = GetSpline3(fXYZ[ind][2], fXYZ[ind + 1][2], fXYZ[ind + 2][2],
                       fXYZ[ind + 3][2], db);
    ind += fNB;
  }
  return {GetSpline3(vx, da), GetSpline3(vy, da), GetSpline3(vz, da)};
}
```

Listing F.10: Scalar spline implementation.

F.2 BENCHMARK IMPLEMENTATION

```
constexpr int NumberOfEvaluations = 10000;
constexpr int FirstMapSize = 4;
constexpr int MaxMapSize = 256;
constexpr int Repetitions = 100;
constexpr auto StepMultiplier = 1.25;

enum EnabledTests {
  Float4,
  Float16,
  Float12,
  Float12Interleaved,
  Horizontal1,
  Horizontal2,
  Alice,
  Autovectorized,
  Scalar,
  NBenchmarks
};

EnabledTests &operator++(EnabledTests &x) {
  return x = static_cast<EnabledTests>(static_cast<int>(x) + 1);
}

template <typename Input> struct VectorizeBuffer {
  typedef simdize<Input> InputV;
  InputV input;
  int entries = 0;
  int operator()(Input x) {
    simdize_assign(input, entries, x);
    entries = (entries + 1) % InputV::size();
    return entries;
  }
};

struct Runner {
  TimeStampCounter tsc;
  double mean[NBenchmarks] = {};
  double stddev[NBenchmarks] = {};
  const std::vector<Point2> &searchPoints;

  Runner(const std::vector<Point2> &s) : searchPoints(s) {}
  template <typename F> void benchmark(const int Test, F &&fun, double err = 20) {
    do {
      mean[Test] = 0;
      stddev[Test] = 0;
      for (const auto &p : searchPoints) {
        fun(p);
      }  // one cache warm-up run to remove one outlier
      for (auto rep = Repetitions; rep; --rep) {
        tsc.start();
        for (const auto &p : searchPoints) {
          fun(p);
        }
        tsc.stop();
        const double x = tsc.cycles() / NumberOfEvaluations;
        mean[Test] += x;
        stddev[Test] += x * x;
      }
      mean[Test] /= Repetitions;
      stddev[Test] /= Repetitions;
      stddev[Test] = std::sqrt(stddev[Test] - mean[Test] * mean[Test]);
    } while (stddev[Test] * err > mean[Test]);
    std::cout << std::setw(9) << std::setprecision(3) << mean[Test];
    std::cout << std::setw(9) << std::setprecision(3) << stddev[Test];
    std::cout << std::flush;
  }
  template <typename I, typename J> void printRatio(I i, J j) {
    const auto ratio = mean[i] / mean[j];
    std::cout << std::setprecision(3) << std::setw(9) << ratio;
    std::cout << std::setprecision(3) << std::setw(9)
              << ratio * std::sqrt(stddev[i] * stddev[i] / (mean[i] * mean[i]) +
                                   stddev[j] * stddev[j] / (mean[j] * mean[j]));
  }
};

int main() {
  std::default_random_engine randomEngine(1);
  std::uniform_real_distribution<float> uniform(-1.f, 1.f);

  std::vector<Point2> searchPoints;
  searchPoints.reserve(NumberOfEvaluations);
  for (int i = 0; i < NumberOfEvaluations; ++i) {
    searchPoints.emplace_back(
        Point2{uniform(randomEngine), uniform(randomEngine)});
  }
  Runner runner(searchPoints);

  for (int MapSize = FirstMapSize; MapSize <= MaxMapSize;
       MapSize *= StepMultiplier) {
    Spline spline(-1.f, 1.f, MapSize, -1.f, 1.f, MapSize);
    Spline2 spline2(-1.f, 1.f, MapSize, -1.f, 1.f, MapSize);
    Spline3 spline3(-1.f, 1.f, MapSize, -1.f, 1.f, MapSize);
    for (int i = 0; i < spline.GetNPoints(); ++i) {
      const float xyz[3] = {uniform(randomEngine), uniform(randomEngine),
                            uniform(randomEngine)};
      spline.Fill(i, xyz);
      spline2.Fill(i, xyz);
      spline3.Fill(i, xyz);
    }
    std::cout << std::setw(8) << spline.GetMapSize() << std::flush;

    for (EnabledTests i = EnabledTests(0); i < NBenchmarks; ++i) {
```

```
    VectorizeBuffer<Point2> vectorizer;
    switch (i) {
    case Scalar:
      runner.benchmark(Scalar, [&] {
        const auto &p2 = spline.GetValueScalar(p);
        asm("" ::"m"(p2));
      });
      break;
    case Alice:
      runner.benchmark(i, [&](const Point2 &p) {
        const auto &p2 = spline.GetValueAlice(p);
        asm("" ::"m"(p2));
      });
      break;
    case Autovectorized:
      runner.benchmark(i, [&](const Point2 &p) {
        const auto &p2 = spline.GetValueAutovec(p);
        asm("" ::"m"(p2));
      });
      break;
    case Float4:
      runner.benchmark(i, [&] {
        const auto &p2 = spline.GetValue(p);
        asm("" ::"m"(p2));
      });
      break;
    case Float16:
      runner.benchmark(i, [&] {
        const auto &p2 = spline.GetValue16(p);
        asm("" ::"m"(p2));
      });
      break;
    case Float12:
      runner.benchmark(i, [&] {
        const auto &p2 = spline2.GetValue(p);
        asm("" ::"m"(p2));
      });
      break;
    case Float12Interleaved:
      runner.benchmark(i, [&] {
        const auto &p2 = spline3.GetValue(p);
        asm("" ::"m"(p2));
      });
      break;
    case Horizontal1:
      runner.benchmark(i, [&] {
        if (0 == vectorizer(p)) {
          const auto &p2 = spline.GetValue(vectorizer.input);
          asm("" ::"m"(p2));
        }
      });
      break;
    case Horizontal2:
      runner.benchmark(i, [&] {
        if (0 == vectorizer(p)) {
          const auto &p2 = spline2.GetValue(vectorizer.input);
          asm("" ::"m"(p2));
        }
      });
    }
    for (EnabledTests i = EnabledTests(0); i < NBenchmarks; ++i) {
      if (i != Scalar) {
        runner.printRatio(Scalar, i);
      }
    }
    std::cout << std::endl;
  }
  return 0;
}
```

Listing F.11: Benchmark program for the different spline implementations.

G

THE SUBSCRIPT OPERATOR TYPE TRAIT

In some situations generic code can base a decision on whether a given type implements the subscript operator or not. This kind of query is part of the type traits that are ubiquitous in generic programming.

The subscript operator poses a special challenge because the non-member subscript operator is not allowed in C++11. The BOOST operator type traits rely on non-member overloads to detect whether a specific operator is callable for a given type. Therefore, the BOOST type traits library does not contain a type trait for the subscript operator.

To implement a type trait the following problem thus needs to be solved: For a given type `T` the expression

```
decltype(std::declval<T>()[0])
```

resolves to the return type of the subscript operator of type `T`. If `T` does not implement a subscript operator that accepts an integer argument, then the expression above is invalid and results in a compilation error. The property, whether a given expression resolves to a valid type or is an invalid expression must be used to create an expression that is either `true` or `false`.

The approach for such issues thus leads us to the SFINAE [81] feature in C++. A function with template parameters, where substitution of the parameters would lead to a substitution failure is not an error, but rather the function will silently be removed from the list of candidates for overload resolution. Thus the function

```
template <typename T,
          typename = decltype(std::declval<T>()[0])> void test();
```

is visible if the subscript operator of `T` exists and invisible otherwise. Thus, we have improved the situation slightly from "compilation error or not" to "function is visible or not". But this is not much better because a function not being visible typically just leads to a compilation error.

The `enable_if` pattern is well known and simply inverts the condition to make a different function visible for the remaining cases:

```
template <typename T>
typename enable_if< condition<T>::value>::type test();
```

```
template <typename T>
typename enable_if<!condition<T>::value>::type test();
```

In that manner we would need an expression that could invert the type of the subscript operator to be there or not. But this is not possible. Thus, there is no way to have one function visible for types with subscript operator and another function visible for types without subscript operator. It is only possible to have one function removed from overload resolution if the subscript operator is not usable.

The solution therefore depends on the parameter to our `test` function. The parameters of a function determine their priority in overload resolution. A function call where the argument type must be implicitly converted is not used if an overload with the exact parameter type is visible. There are many possible types to use here, but one of the simpler cases is the constant `1` which is of the exact type **`int`**, but can also be implicitly converted to **`float`**. The two functions thus become:

```
template <typename T,
          typename = decltype(std::declval<T>()[0])> void test(int);
template <typename T> void test(float);
```

Finally, the selected function must lead to a `true` or `false` value. This is easy with the **`std::true_type`** and **`std::false_type`** types. The test functions can use these types as return types. Thus, the complete type trait can be implemented as:

```
template <typename T,
          typename =
              decltype(std::declval<T>()[0])> std::true_type test(int);
template <typename T> std::false_type test(float);

template <typename T>
struct has_subscript_operator : public decltype(test<T>(1)) {};
```

Now, the constant expression `has_subscript_operator<T>::value` is a boolean value that can be used to, for instance, base implementation strategies on the presence of a subscript operator of a template parameter type **`T`**.

H

THE ADAPTSUBSCRIPTOPERATOR CLASS

```
template <typename Base> class AdaptSubscriptOperator : public Base {
public:
  using Base::Base;

  // explicitly enable Base::operator[] because the following would hide it
  using Base::operator[];

  // forward to non-member subscript_operator function
  template <
      typename I,
      typename = typename std::enable_if<!std::is_arithmetic<typename std::decay<
          I>::type>::value>::type  // arithmetic types should always use
                                   // Base::operator[] and never match this one
      >
  auto operator[](I &&arg)
      -> decltype(subscript_operator(*this, std::forward<I>(arg))) {
    return subscript_operator(*this, std::forward<I>(arg));
  }

  // const overload of the above
  template <typename I,
            typename = typename std::enable_if<
                !std::is_arithmetic<typename std::decay<I>::type>::value>::type>
  auto operator[](I &&arg) const
      -> decltype(subscript_operator(*this, std::forward<I>(arg))) {
    return subscript_operator(*this, std::forward<I>(arg));
  }
};
```

Listing H.1: Generic adaptor class to add the forwarding subscript operator to existing container classes. This enables non-member subscript functionality for the adapted class.

I

SUPPORTING CUSTOM DIAGNOSTICS AND SFINAE

The following is the reproduction of a paper that was submitted to the C++ committee and discussed at the Urbana 2014 meeting. It is relevant for the interface design described in Section 4.5.

Committee feedback was that there is general interest in solving the problem. But at this point it is not clear enough (to the committee) whether the improved diagnostics from concepts [77] will already solve the issue in an acceptable manner. To go forward this paper needs more motivational examples and possibly a more general solution.

I.1 PROBLEM

Static assertions are a very useful tool to improve error messages if a library interface is used incorrectly. Consider the addition operator in Listing I.1.

This has the following effects:

1. `operator+` is a *viable* function for portable and unportable uses of the addition operator.
2. The program is *ill-formed* if an unportable type combination is used.
3. The compiler will output the second argument to the `static_assert` as *custom diagnostic* output if an unportable type combination is used.

```
class simd_float;
template <typename T> simd_float operator+(simd_float, T) {
  static_assert(has_compatible_vector_size<simd_float, T>::value,
                "Incompatible operands: the SIMD register sizes for "
                "both operands must be equal on all possible target "
                "platforms to ensure portable code. Use an explicit "
                "type conversion to make the code portable.");
  return ...;
}
```

Listing I.1: Example usage of `static_assert` for a more informative error message.

```
template <typename T, typename U,
          typename = decltype(std::declval<T>() + std::declval<U>())>
std::true_type test(int);
template <typename T, typename U> std::false_type test(...);
template <typename T, typename U = T>
struct has_addition_operator : public decltype(test<T, U>(1)) {};
```

Listing I.2: A type trait that checks for the existence of `operator+(T, U)`.

```
template <typename T>
enable_if_t<has_compatible_vector_size<simd_float, T>::value,
            simd_float>
    operator+(simd_float, T);
template <typename T>
enable_if_t<!has_compatible_vector_size<simd_float, T>::value,
            simd_float>
    operator+(simd_float, T) = delete;
```

Listing I.3: Using a deleted function as an alternative implementation to Listing I.1.

4. A *SFINAE* (or concept) check for the *usability* of the addition operator for an unportable type combination is impossible to implement.

The **`has_addition_operator`** trait in Listing I.2 will not tell whether a call to `operator+(T, U)` leads to a failed static assertion. This depends on the rules of substitution failure: The expression `decltype(std::declval<T>() + std::declval<U>())>` yields a valid type even if `has_compatible_vector_size<simd_float, T>::value` is `false`.[1] The substitution rules do not depend on whether a static assertion fails on instantiation of a template function. They do depend on whether the (viable) function is accessible (`public` vs. `private`) or *deleted*, though. Thus, the **`has_addition_operator`** trait will tell whether an addition operator is inaccessible or deleted.

Listing I.3 shows an implementation of `operator+` that solves the SFINAE issue of Listing I.1 but at the cost of losing custom diagnostics output. The compiler has no idea why the library developer decided to declare the function as deleted. Thus, all it can do is tell that a deleted function was used. This tells a user of the library that either the library developer made a mistake or it was really intended that this overload is forbidden.

There is no way in current C++ to declare a function in such a way that all four items are satisfied:

1. *viable* for incorrect use

2. *ill-formed* for incorrect use

3. *custom diagnostics* output for incorrect use

1 It would be possible to modify Listing I.1 such that the return type is invalid, but then the function would not be viable for unportable type combinations and the `static_assert` would never fail…

4. *SFINAE* or Concepts can check for *usability*, not only viability

The *Custom diagnostics* and *SFINAE* features are mutually exclusive.

I.2 POSSIBLE SOLUTIONS

Approaches:

1. Introduce a new type trait (which requires compiler support) that can detect whether a given expression fails a static assertion.

2. Extend concepts to do "negative matching" to enable customized diagnostics. Thus, a call to `simd_float + double` would match the second overload in Listing I.4 as best viable function and make such a program ill-formed with the string after `error` used for diagnostics. In a template parameter substitution this would lead to a failure and thus enable `has_addition_operator` to check for usability of the addition operator.

```
template <typename T>
  requires has_compatible_vector_size<simd_float, T>::value
simd_float operator+(simd_float, T);
template <typename T>
  requires !has_compatible_vector_size<simd_float, T>::value
  error "<how to use + correctly>"
simd_float operator+(simd_float, T);
```

Listing I.4: Notion of "negative matching" as an extension to concepts.

3. Introduce an additional check at the end of overload resolution [16, §13.3], in the same spirit as the check for accessibility:

§13.3 [over.match]

If a best viable function exists and is unique, overload resolution succeeds and produces it as the result. Otherwise overload resolution fails and the invocation is ill-formed. When overload resolution succeeds, and the best viable function is not accessible (Clause 11) in the context in which it is used or template instantiation would lead to a failed `static_assert`, the program is ill-formed.

The intention is to trigger a substitution failure when a `static_assert` would fail and thus enable SFINAE.

4. Extend the `delete` expression for deleted functions [16, §8.4.3] to accept an optional string argument that will be used for diagnostics output.

§8.4.1 [dcl.fct.def.general]

Function definitions have the form

function-definition:

attribute-specifier-seq$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator virt-specifier-seq*$_{opt}$ *function-body*

function-body:

```cpp
template <typename T>
enable_if_t<!has_compatible_vector_size<simd_float, T>::value,
            simd_float>
    operator+(simd_float, T) =
        delete ("Incompatible operands: the SIMD register sizes for "
                "both operands must be equal on all possible target "
                "platforms to ensure portable code. Use an explicit "
                "type conversion to make the code portable.");
```

Listing I.5: Providing custom diagnostics to a deleted function.

ctor-initializer$_{opt}$ *compound-statement*
function-try-block
deleted-definition
`= default ;`
~~`= delete ;`~~

deleted-definition:
`= delete (` *string-literal* `) ;`
`= delete ;`

§8.4.3 [dcl.fct.def.delete]

A function definition of the form:
attribute-specifier-seq$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator* *virt-specifier-seq*$_{opt}$ ~~`= delete ;`~~ *deleted-definition*
is called a *deleted definition*. A function with a deleted definition is also called a *deleted function*.

A program that refers to a deleted function implicitly or explicitly, other than to declare it, is ill-formed, and the resulting diagnostic message (1.4) shall include the text of the string-literal, if one is given, except that characters not in the basic source character set (2.3) are not required to appear in the diagnostic message. [*Note:* This includes calling the function implicitly or explicitly and forming a pointer or pointer-to-member to the function. It applies even for references in expressions that are not potentially-evaluated. If a function is overloaded, it is referenced only if the function is selected by overload resolution. — *end note*]

With this solution the deleted function in Listing I.3 can be extended as shown in Listing I.5.

I.3 EVALUATION

Approaches 1 and 3 require instantiation of the constant part of the function body to evaluate the `static_assert` during overload resolution, before actually selecting the function. This seems novel territory for such a small feature.

Similarly, approach 2 requires extensions to the concepts design, which is currently not even a working draft for a Technical Specification.

Approach 4 can be implemented as a fairly small extension to the current check whether a function is deleted at the end of overload resolution. The issue of integrating string-literals from program source code to diagnostic compiler output was already solved for `static_assert`.

The recommendation is to proceed with approach 4.

ACRONYMS

ABI	Application Binary Interface
ALICE	A Large Ion Collider Experiment
ALU	Arithmetic Logic Unit
API	official: Application Programming Interface better: Application Programmer Interface
AVX	Advanced Vector Extensions
CBM	Compressed Baryonic Matter; experiment at FAIR
CERN	European Organization for Nuclear Research; in Geneva
CPU	Central Processing Unit
FAIR	Facility for Antiproton and Ion Research; in Darmstadt
FLOP	floating-point operation
FMA	Fused Multiply-Add
GCC	GNU Compiler Collection
Geant	GEometry ANd Tracking
Geant-V	GEometry ANd Tracking Vector (Prototype) project
GPU	Graphics Processing Unit
HLT	High-Level Trigger
ICC	Intel C++ Compiler
ILP	Instruction Level Parallelism
ITS	Inner Tracking System
LHC	Large Hadron Collider
LRU	Least Recently Used

LTO	Link Time Optimization
MIC	Many Integrated Core
MMX	Multimedia Extensions
PE	Processing Element
POD	Plain Old Data
RHIC	Relativistic Heavy Ion Collider; at Brookhaven National Laboratory
SFINAE	Substitution Failure Is Not An Error
SIMD	Single Instruction, Multiple Data
SISD	Single Instruction, Single Data
SSE	Streaming SIMD Extensions
STAR	Solenoidal Tracker at RHIC
STL	Standard Template Library
TBB	Intel Threading Building Blocks
TLB	translation look aside buffer
TPC	Time Projection Chamber
TU	translation unit

BIBLIOGRAPHY

[1] *About | ROOT*. The ROOT Team. URL: `https://root.cern.ch/drupal/content/about` (visited on 02/21/2015).

[2] T. Alt et al. "The ALICE high level trigger." In: *Journal of Physics G: Nuclear and Particle Physics* 30.8 (2004), S1097. DOI: `10.1088/0954-3899/30/8/066`.

[3] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities." In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: `10.1145/1465482.1465560`.

[4] *Automatic Vectorization*. Intel Corporation. 2013. URL: `http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/GUID-7D541D6D-4929-4F35-A58D-B67F9A897AA0.htm` (visited on 03/20/2013).

[5] *Auto-vectorization in GCC*. English. Free Software Foundation. 2012. URL: `http://gcc.gnu.org/projects/tree-ssa/vectorization.html` (visited on 03/20/2013).

[6] Jon Louis Bentley. "Multidimensional Binary Search Trees Used for Associative Searching." In: *Communications of the ACM* 18.9 (Sept. 1975), pp. 509–517. ISSN: 0001-0782. DOI: `10.1145/361002.361007`.

[7] Dirk Brandewie and Rafael J. Wysocki. *Intel P-state driver*. 2014. URL: `https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt` (visited on 02/04/2015).

[8] Rene Brun and Fons Rademakers. "ROOT — An object oriented data analysis framework." In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 389.1-2 (Apr. 1997), pp. 81–86. ISSN: 01689002. DOI: `10.1016/S0168-9002(97)00048-X`.

[9] Oliver Simon Brüning et al. *LHC Design Report*. Geneva: CERN, 2004.

[10] Federico Carminati et al. "A concurrent vector-based steering framework for particle transport." In: *Journal of Physics: Conference Series* 523.1 (2014). DOI: `10.1088/1742-6596/523/1/012004`.

[11] Federico Carminati et al. "The Path toward HEP High Performance Computing." In: *Journal of Physics: Conference Series* 513.5 (2014). DOI: 10.1088/1742-6596/513/5/052006.

[12] ALICE Collaboration. *Technical proposal for A Large Ion Collider Experiment at the CERN LHC*. Tech. rep. CERN, Dec. 1995.

[13] ALICE HLT Collaboration. *ALICE High-Level Trigger Conceptual Design*. Tech. rep. CERN, Dec. 2002.

[14] The ALICE Collaboration et al. "The ALICE experiment at the CERN LHC." In: *Journal of Instrumentation* 3.08 (2008), S08002. DOI: 10.1088/1748-0221/3/08/S08002.

[15] *CUDA C Programming Guide. Design Guide*. NVIDIA. Aug. 2014. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (visited on 01/22/2015).

[16] Stefanus Du Toit, ed. *Working Draft, Standard for Programming Language C++*. ISO/IEC C++ Standards Committee Paper. N3936. 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/prot/14882fdis/n3936.pdf.

[17] Pierre Estérie et al. "Boost.SIMD: Generic Programming for Portable SIMDization." In: *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*. WPMVP '14. New York, NY, USA: ACM, 2014, pp. 1–8. ISBN: 978-1-4503-2653-7. DOI: 10.1145/2568058.2568063.

[18] Pierre Estérie et al. "Exploiting Multimedia Extensions in C++: A Portable Approach." In: *Computing in Science & Engineering* 14.5 (Sept. 2012), pp. 72–77. ISSN: 1521-9615. DOI: 10.1109/MCSE.2012.96. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6320573.

[19] Christian Wolfgang Fabjan et al. *ALICE trigger data-acquisition high-level trigger and control system: Technical Design Report*. Technical Design Report ALICE. Geneva: CERN, 2004.

[20] Joel Falcou. *NT2 : The Numerical Template Toolbox*. 2007. URL: http://nt2.sourceforge.net/ (visited on 10/19/2011).

[21] Joel Falcou and Jocelyn Serot. "E.V.E., An Object Oriented SIMD Library." In: *Scalable Computing: Practice and Experience* 6.4 (2005), pp. 31–41.

[22] R.A. Finkel and J.L. Bentley. "Quad trees a data structure for retrieval on composite keys." In: *Acta Informatica* 4 (1974), pp. 1–9. ISSN: 0001-5903. DOI: 10.1007/BF00288933.

[23] Yuri Fisyak et al. *Track Reconstruction in the STAR TPC with a CA Based Approach*. Tech. rep. GSI, 2010.

[24] M. Flynn. "Some Computer Organizations and Their Effectiveness." In: *Computers, IEEE Transactions on* C-21.9 (1972), pp. 948–960. ISSN: 0018-9340. DOI: 10.1109/TC.1972.5009071.

[25] Agner Fog. *C++ vector class library*. 2014. URL: http://www.agner.org/optimize/vectorclass.pdf.

[26] Agner Fog. *Instruction tables*. Manual. Copenhagen: Copenhagen University College of Engineering, 2012. URL: http://www.agner.org/optimize/.

[27] Tim Foley and Jeremy Sugerman. "KD-tree Acceleration Structures for a GPU Raytracer." In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '05. New York, NY, USA: ACM, 2005, pp. 15–22. ISBN: 1-59593-086-8. DOI: 10.1145/1071866.1071869.

[28] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. "An Algorithm for Finding Best Matches in Logarithmic Expected Time." In: *ACM Trans. Math. Softw.* 3.3 (1977), pp. 209–226. ISSN: 0098-3500. DOI: 10.1145/355744.355745.

[29] Rudolf Frühwirth. *Data analysis techniques for high-energy physics*. 2nd ed. Cambridge: Univ. Pr., 2000. ISBN: 9780521635486.

[30] Robert Geva and Clark Nelson. *Language Extensions for Vector level parallelism*. ISO/IEC C++ Standards Committee Paper. N3831. 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3831.pdf.

[31] Andrei Gheata et al. "Rethinking particle transport in the many-core era towards GEANT 5." In: *Journal of Physics: Conference Series* 396.2 (2012). DOI: 10.1088/1742-6596/396/2/022014.

[32] Sergey Gorbunov, Matthias Kretz, and David Rohr. *Fast Cellular Automaton tracker for the ALICE High Level Trigger*. Tech. rep. GSI, 2009, p. 348.

[33] Sergey Gorbunov et al. "Fast SIMDized Kalman filter based track fit." In: *Computer Physics Communications* 178 (2008), pp. 374–383.

[34] S. Gorbunov et al. "ALICE HLT High Speed Tracking on GPU." In: *Nuclear Science, IEEE Transactions on* 58.4 (Aug. 2011), pp. 1845–1851. ISSN: 0018-9499. DOI: 10.1109/TNS.2011.2157702.

[35] Pablo Halpern. *An Abstract Model of Vector Parallelism*. ISO/IEC C++ Standards Committee Paper. N4238. 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4238.pdf.

[36] Jared Hoberock. *Working Draft, Technical Specification for C++ Extensions for Parallelism*. ISO/IEC C++ Standards Committee Paper. N4071. 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4071.htm.

[37] Daniel Reiter Horn et al. "Interactive k-d Tree GPU Raytracing." In: *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. ACM, 2007, pp. 167–174. DOI: 10.1145/1230100.1230129.

[38] *ILLIAC IV Systems Characteristics and Programming Manual*. Burroughs Corporation. May 1972. 364 pp.

[39] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation. Jan. 2011. URL: http://www.intel.com/products/processor/manuals/.

[40] *Intel ® 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation. Sept. 2014. URL: http://www.intel.com/products/processor/manuals/.

[41] *Intel ® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. June 2013. URL: http://www.intel.com/products/processor/manuals/.

[42] *Intel® C++ Compiler XE 13.1 User and Reference Guides — simd*. Intel Corporation. Dec. 2013. URL: https://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-1EA04294-988E-4152-B584-B028FD6FAC48.htm (visited on 12/10/2014).

[43] *Intel® Cilk™ Plus Language Extension Specification*. Intel Corporation. Sept. 2013. URL: https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm.

[44] *Intel® Cilk™ Plus Language Extension Specification Version 1.2*. Intel Corporation. Sept. 2013. URL: http://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm (visited on 12/09/2014).

[45] *Intel® Threading Building Blocks*. Intel Corporation. Aug. 2011. 381 pp. URL: http://threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Reference.pdf (visited on 10/20/2011).

[46] *Intel® Xeon Phi™ Coprocessor Instruction Set Architecture Reference Manual*. Intel Corporation. 2012.

[47] *Introducing NEON™*. Development Article. ARM Limited, 2009. 18 pp. URL: http://infocenter.arm.com/help/topic/com.arm.doc.dht0002a/DHT0002A_introducing_neon.pdf.

[48] *ISO/IEC 14882:2011. Information technology — Programming languages — C++*. Standard. ISO/IEC JTC 1/SC 22, Sept. 1, 2011. URL: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=50372.

[49] *ISO/IEC PDTS 19570. C++ Extensions for Parallelism*. Standard Draft. ISO/IEC JTC 1/SC 22, 2014. URL: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=65241.

[50] *Itanium C++ ABI*. CodeSourcery et al. URL: http://www.codesourcery.com/cxx-abi/ (visited on 12/12/2014).

[51] R.E. Kalman. "A new approach to linear filtering and prediction problems." In: *Journal of Basic Engineering*. Vol. 82. 1960, pp. 35–45.

[52] I. Kisel, I. Kulakov, and M. Zyzak. "Standalone First Level Event Selection Package for the CBM Experiment." In: *Nuclear Science, IEEE Transactions on* 60.5 (Oct. 2013), pp. 3703–3708. ISSN: 0018-9499. DOI: 10.1109/TNS.2013.2265276.

[53] Ivan Kisel, Matthias Kretz, and Igor Kulakov. *Scalability of the SIMD Kalman Filter Track Fit Based on the Vector Classes*. Tech. rep. GSI, 2009, p. 56.

[54] Donald E. Knuth. *The Art of Computer Programming 3. Sorting and Searching*. 2nd. Vol. 3. Addison Wesley, 1998, p. 780. ISBN: 978-0201896855.

[55] Matthias Kretz. "Efficient Use of Multi- and Many-Core Systems with Vectorization and Multithreading." Diplomarbeit. University of Heidelberg, Dec. 2009.

[56] Matthias Kretz. *SIMD Types: The Mask Type & Write-Masking*. ISO/IEC C++ Standards Committee Paper. N4185. 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4185.pdf.

[57] Matthias Kretz. *SIMD Types: The Vector Type & Operations*. ISO/IEC C++ Standards Committee Paper. N4184. 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4184.pdf.

[58] Matthias Kretz. *SIMD Vector Types*. ISO/IEC C++ Standards Committee Paper. N3759. 2013. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3759.html.

[59] Matthias Kretz and Volker Lindenstruth. "Vc: A C++ library for explicit vectorization." In: *Software: Practice and Experience* (2011). ISSN: 1097-024X. DOI: 10.1002/spe.1149.

[60] Matthias Kretz and Jens Maurer. *Supporting Custom Diagnostics and SFINAE*. ISO/IEC C++ Standards Committee Paper. N4186. 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4186.pdf.

[61] Victor Luchangco, Jens Maurer, Michael Wong, et al. *Standard Wording for Transactional Memory Support for C++*. ISO/IEC C++ Standards Committee Paper. N3999. 2014. URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3999.pdf`.

[62] *macstl*. Pixelglow Software. Sept. 2005. URL: `http://www.pixelglow.com/macstl/` (visited on 01/25/2011).

[63] *Malloc-Examples*. Free Software Foundation. URL: `http://www.gnu.org/software/libc/manual/html_node/Malloc-Examples.html` (visited on 01/05/2015).

[64] Michael Matz et al. *System V Application Binary Interface*. Nov. 2014. URL: `http://www.x86-64.org/documentation_folder/abi-0.99.pdf`.

[65] Aaftab Munshi, ed. *The OpenCL Specification*. Specification. Version 1.2. Khronos Group, Nov. 14, 2012. URL: `https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`.

[66] J von Neumann. "First Draft of a Report on the EDVAC." In: *Annals of the History of Computing, IEEE* (1993). URL: `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=238389`.

[67] Chris J. Newburn et al. "Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language." In: *Proceedings - International Symposium on Code Generation and Optimization, CGO 2011*. 2011, pp. 224–235.

[68] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.0*. July 2013. URL: `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[69] *Pentium® Processor with MMX™ Technology*. Datasheet. Order Number: 243185.004. Intel Corporation. June 1997. 51 pp. URL: `http://download.intel.com/design/archives/processors/mmx/docs/24318504.pdf` (visited on 07/05/2013).

[70] *PowerPC Microprocessor Family : Vector / SIMD Multimedia Extension Technology Programming Environments Manual*. 2.07c. IBM Corporation. 2006. 329 pp. URL: `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/6B4C13893761AAB885257B030066C0D9`.

[71] Torvald Riegel. *Light-Weight Execution Agents Revision 3*. ISO/IEC C++ Standards Committee Paper. N4156. 2014. URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4156.pdf`.

[72] David Rohr. "On development, feasibility, and limits of highly efficient CPU and GPU programs in several fields." PhD thesis. 2014, p. 254. eprint: `urn:nbn:de:hebis:30:3-343772`.

[73] Jeff Snyder and Chandler Carruth. *Call for Compile-Time Reflection Proposals*. ISO/IEC C++ Standards Committee Paper. N3814. 2013. URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3814.html`.

[74] *Software Optimization Guide for AMD Family 10h and 12h Processors*. Publication # 40546. Advanced Micro Devices. Dec. 2010. 346 pp. URL: `http://developer.amd.com/documentation/guides/Pages/default.aspx` (visited on 01/25/2011).

[75] Bjarne Stroustrup. *Stroustrup: C++ Style and Technique FAQ*. 2013. URL: `http://www.stroustrup.com/bs_faq2.html#overload-dot` (visited on 05/08/2013).

[76] Robert Strzodka. "GPU Computing Gems Jade Edition." In: *GPU Computing Gems Jade Edition*. Elsevier, 2012, pp. 429–441. ISBN: 9780123859631. DOI: `10.1016/B978-0-12-385963-1.00031-9`.

[77] Andrew Sutton, Bjarne Stroustrup, and Gabriel Dos Reis. *Concepts Lite Specification*. ISO/IEC C++ Standards Committee Paper. N3819. 2013. URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3819.pdf`.

[78] Andrew S. Tanenbaum. *Structured Computer Organization*. Upper Saddle River, NJ: Prentice Hall, 1999, p. 669. ISBN: 0-13-020435-8.

[79] *The Open Group Base Specifications, Issue 6, IEEE Std 1003.1*. Specification. IEEE and The Open Group, 2004. URL: `http://pubs.opengroup.org/onlinepubs/009695399/`.

[80] *Tutorial: Array Notation | Cilk Plus*. Intel Corporation. URL: `https://www.cilkplus.org/tutorial-array-notation` (visited on 01/11/2014).

[81] David Vandevoorde and Nicolai M Josuttis. *C++ Templates : The Complete Guide*. Boston, MA: Addison-Wesley, 2003, p. 528. ISBN: 0201734842. URL: `http://proquest.safaribooksonline.com/0201734842`.

[82] Todd Veldhuizen. "Expression Templates." In: *C++ Report* 7.5 (1995), pp. 26–31.

[83] Haichuan Wang et al. "Simple, Portable and Fast SIMD Intrinsic Programming: Generic Simd Library." In: *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*. WPMVP '14. New York, NY, USA: ACM, 2014, pp. 9–16. ISBN: 978-1-4503-2653-7. DOI: `10.1145/2568058.2568059`.

[84] Sandro Wenzel et al. "Vectorising the detector geometry to optimise particle transport." In: *Journal of Physics: Conference Series* 513.5 (2014). DOI: `10.1088/1742-6596/513/5/052038`.

[85] Sergey Zubkov. *std::valarray::operator=*. 2014. URL: `http://en.cppreference.com/w/cpp/numeric/valarray/operator=` (visited on 12/15/2014).

INDEX

?:, 84
$\mathcal{W}_{\mathtt{T}}$, 4, 109
$\mathcal{S}_{\mathtt{T}}$, 4

ABI, 20, 99
aliasing, 19
ALICE, 155
alignment, 7, 20, 42
AltiVec, 8
ALU, 5
Amdahl's law, 151, 164, 165, 171
AoS, 113, 114, 126, 176, 239
AoVS, 113, 114, 139, 167, 239
array machine, 5
array notation, 55, 74
auto-vectorization, 15, 16, 18, 19, 23, 130, 161, 164, 166
AVX, 4, 9

benchmark, 160
binary operators, 45
bitwise operators, 81
branch prediction, 11
branching, 18, 164
broadcast, 37
builtin vector types, 24

cache, 151, 160, 161, 164
CBM, 166
Cilk Plus, 23, 55, 56, 74, 84
clang, 24
comparison operators, 81
complexity, 151, 160
compound assignment, 50
conditional operator, 84
container, 111
conversion, 38, 95
converting loads, 41
converting stores, 41
coordinate transformation, 156
countable, 15, 18

data-parallelism, 3
default vector type, 68
deinterleave, 160, 164

EntryReference, 33, 78
EntryType, 33, 78
exception, 21, 75, 110
expression templates, 96

FMA, 4
for_each, 110

gather, 6, 7, 9, 52, 156, 160, 164, 165
GCC, 16, 24, 131, 161
Geant-V, 167

HLT, 155
horizontal vectorization, 114, 117, 156, 159, 164, 165

iif, 84
ILP, 11, 96, 134
implicit masking, 75
IndexesFromZero(), 34
internal data, 70
intrinsics, 24
ITS, 155

Kalman-filter, 85, 166
KF Particle, 167

LHC, 155

load, 6, 40, 109
loop-vectorization, 15, 16, 21
LTO, 20

mask initialization, 79
mask load/store, 80
mask type, 76, 98
Mask<T>, 77
masked gather, 55, 87
masked scatter, 55, 87
MaskType, 33
may_alias, 100
MemoryAlignment, 34
MIC, 54
multi-threading, 12

NEON, 8
non-temporal access, 42, 55

OpenMP, 12, 23
operator.(), 67

padding, 157
parallel algorithms, 109
PE, 5
pipelining, 11
POD, 35, 57
pointer chasing, 134, 149
portability, 158
POSIX scatter/gather, 53
prefetch, 43

reduction, 82, 131, 137, 147
  all_of, 82
  any_of, 82, 131
  none_of, 82
  some_of, 82
register pressure, 96

scalar implementation, 68
scatter, 52
serial semantics, 16
SFINAE, 45, 122, 213, 217–219
shuffle, 6
SIMD, 3, 5
SIMD lane, 5
simd_cast, 39, 40, 159
SimdArray, 94, 158, 160
simdize, 116
simdize_extract, 123
simdize_insert, 123
SISD, 5
size(), 34
SoA, 113, 114, 126, 176, 239
speculative execution, 11
spline, 156
SSE, 4, 9
STAR, 166
std::par_vec, 109
STL, 93
store, 40, 109
streaming store, 43
subscript operator, 51, 57
superscalar execution, 11

TBB, 12
TPC, 155
track reconstruction, 155, 166
transpose, 159, 164
TU, 100
tuple_size, 123

unary operators, 44
union, 100
unrolling, 11

valarray, 53, 55, 56, 84, 85, 93–95
  indirect_array, 55
Vc::where, 86
vector initialization, 35
vector lane, 5
vector machine, 5
vector semantics, 17, 110

`Vector<bool>`, 77
vectorization, 15, 113
`VectorType`, 33, 78
vertical vectorization, 11, 134, 156, 157, 160, 164, 165
Von Neumann architecture, 5

working set, 12, 13, 95
write-masking, 84, 85

Xeon Phi, 9

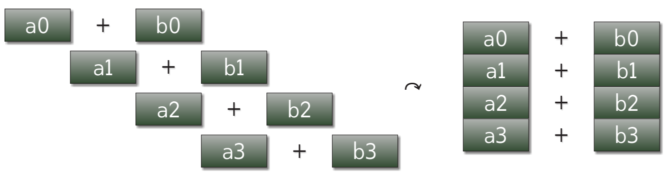
GERMAN SUMMARY

Datenparalleles Programmieren ist wichtiger als jemals zuvor, denn die serielle Leistung stagniert. Alle wichtigen Computerarchitekturen haben ihre Unterstützung für universales Rechnen in expliziter datenparalleler Ausführung erweitert und weitere Erweiterungen sind angekündigt. Diese Fortschritte paralleler Hardware wurden allerdings nicht von den entsprechend nötigen Erweiterungen der etablierten Programmiersprachen begleitet. Softwareentwickler wurden also nicht in die Lage versetzt die inhärente Datenparallelität ihrer Algorithmen explizit anzugeben. CPU und GPU Hersteller haben daher neue Sprachen, Spracherweiterungen oder Dialekte entwickelt, um datenparalleles Programmieren zu ermöglichen. Diese Lösungen haben allerdings einige Nachteile und sind nicht für jede Aufgabe das geeignete Werkzeug. Die vorliegende Dissertation befasst sich daher mit einer datentypbasierten Lösung für die C++ Programmiersprache.

SIMD

Die Idee der datenparallelen Ausführung rührt von der Beobachtung her, dass in rechenintensiven Algorithmen häufig verschiedene Daten mit den gleichen Operationen verarbeitet werden. Daraus folgte die Entwicklung von Computern, die mit einer einzelnen Instruktion mehrere Daten verarbeiten konnten (Flynn [3] prägte dafür den Namen *SIMD*[2]). Softwareseitig wird die bestehende Datenparallelität in Algorithmen aber gar nicht als parallele Ausführung abgebildet, denn typischerweise bieten Programmiersprachen für das Verarbeiten von unterschiedlichen Daten mit den gleichen Operationen nur Schleifenausdrücke an. Schleifen sind aber immer als serielle Iterationen über eine Iterationsvariable definiert. Damit liegt der Sprache inhärent eine serielle Semantik zugrunde, welche in *ISO/IEC 14882:2011* [9, §1.9 p13] durch die *sequenced before* Regel charakterisiert ist.

Das folgende Bild zeigt die konzeptionelle Transformation, die nötig ist, um von der seriellen Ausführung von vier Additionen zu der parallelen Ausführung zu gelangen:



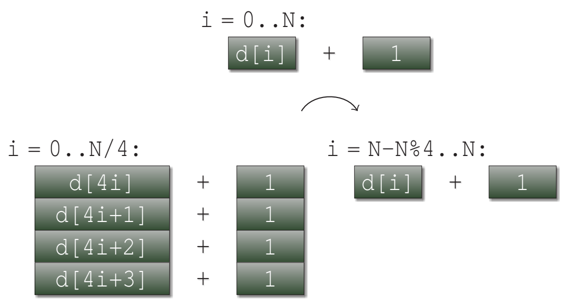
2 Single Instruction, Multiple Data

Beide Varianten sind heutzutage auf einem Hauptprozessor ausführbar. Die linke Variante braucht vier Instruktionen während die Rechte mit einer einzigen SIMD Instruktion auskommt. Da auf aktuellen Prozessoren die Ausführungskosten für eine skalare Instruktion weitestgehend mit denen einer SIMD Instruktion gleich sind, ist die rechte Variante vier mal effizienter.

VEKTORISIERUNG

Diese Transformation nennt man auch *Vektorisierung*, da skalare Variablen (skalare Register) in Vektorvariablen (Vektorregister) transformiert werden. Die folgende Illustration zeigt die Transformation einer Schleife, welche von einem automatischen Vektorisierer (Autovektorisierung) durchgeführt wird:

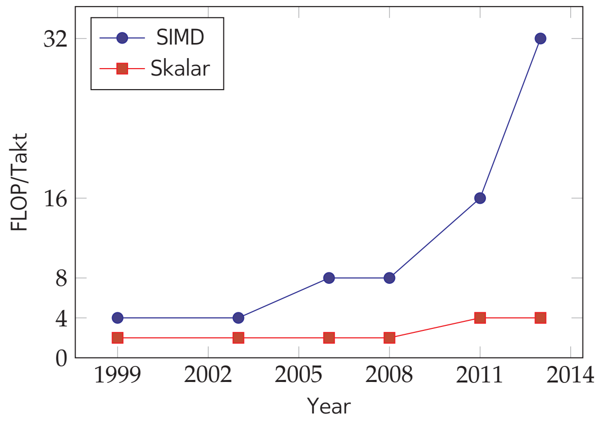
```
void func(float *d) {
  for (int i = 0; i < N; ++i) {
    d[i] += 1.f;
  }
}
```



Dabei wird eine Schleife über `N` Additionen in eine vektorisierte Schleife, und eventuell skalare Prolog- und Epilogschleifen, übersetzt. Dies ist die grundlegende Strategie für automatische Vektorisierung der meisten datenparallelen Probleme. Bei einer manuellen Vektorisierung geht der Softwareentwickler meist analog vor, da der Großteil der Datenparallelität in skalaren Quelltexten in Schleifen abgebildet ist. Dabei hat der Softwareentwickler aber zusätzlich die Möglichkeit Datenstrukturen anzupassen, was einem Compiler, zumindest mit C/C++, nicht erlaubt ist.

Nun könnte man durchaus die Auffassung vertreten, dass automatisches Vektorisieren gut genug funktioniere und manuelles Vektorisieren zu aufwändig sei. Allerdings sollte man bei dieser Überlegung beachten wie viel effizienter eine vektorisierte Verarbeitung sein kann, und ob es im Zweifelsfall vertretbar ist darauf zu verzichten. Denn meist erfordert die automatische Vektorisierung auch manuelle Anstrengungen und bewusste Einschränkungen beim Entwickeln.

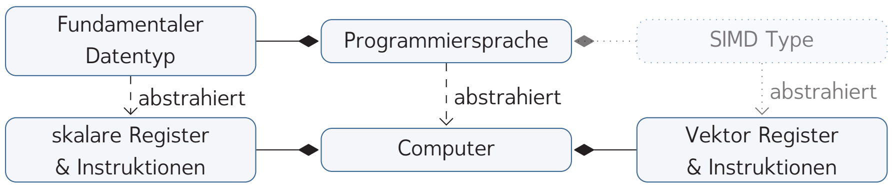
Die folgende Abbildung zeigt die maximale Anzahl von Fließkommaoperationen (FLOP) in einfacher Genauigkeit pro Takt für SIMD Instruktionen und skalare Instruktionen für x86 Prozessoren (Maximum von AMD und Intel) aufgetragen über die Jahre der Veröffentlichung einer neuen Mikroarchitektur [4, 8, 10]. Dabei wird nur ein einziger Prozessorkern (bzw. Thread) berücksichtigt:



Da eine Anwendung, welche viel Rechenzeit beansprucht, viel Geld sparen kann wenn sie effizienter rechnet (oder weniger Energie verbraucht wenn sie schneller wieder in den Leerlauf zurückkehrt), dürfen heutzutage die wenigsten Entwickler die mögliche Effizienzsteigerung durch SIMD ignorieren.

SIMD DATENTYPEN

Die folgende Abbildung zeigt wie Programmiersprachen die Funktionsweise eines Computers abstrahieren:



Der Prozessor des Computers wird über Register und Instruktionen gesteuert. Die meisten Sprachen, insbesondere C++, benutzen Datentypen und deren zugehörige Operatoren, um Register und Instruktionen zu abstrahieren. Allerdings existiert seit der Einführung von SIMD Registern und Instruktionen eine Diskrepanz in der Abstraktion.

ARITHMETISCHE TYPEN

Daher ist es sinnvoll neue Datentypen zu definieren, welche die Vektor Register und Instruktionen abbilden. Das Klassen *template* `Vector<T>` soll entsprechend die folgenden Eigenschaften haben:

- Der Wert eines Objekts vom Typ `Vector<T>` besteht aus $\mathcal{W}_T$ skalaren Werten vom Typ `T`.
- $\mathcal{W}_T$, `sizeof` und `alignof` von `Vector<T>` sind abhängig vom Zielrechner.

- Die skalare Einträge des Objekts lassen sich als *lvalue* Referenz abrufen.
- Die Anzahl der skalaren Einträge ($\mathcal{W}_\mathtt{T}$) ist als konstanter Ausdruck bereitgestellt.
- Operatoren die mit `T` funktionieren, lassen sich mit der gleichen Semantik pro Eintrag auf `Vector<T>` anwenden.
- Das Ergebnis jedes skalaren Wertes einer Operation auf `Vector<T>` ist unabhängig von $\mathcal{W}_\mathtt{T}$.
- Die Syntax und Semantik der fundamentalen arithmetischen Datentypen lassen sich direkt auf `Vector<T>` übertragen.
- Der Compiler ist fähig Optimierungsschritte genauso anzuwenden wie auf skalare Operationen.

MASKEN

Äquivalent zu `Vector<T>` wird ein Klassen *template* `Mask<T>` definiert. Diese Datentypen werden für den Rückgabewert von Vergleichsoperatoren von `Vector<T>` benötigt. Sie identifizieren $\mathcal{W}_\mathtt{T}$ boolesche Werte.

Des Weiteren sind Maskentypen wichtig, um bedingte Ausführung zu vektorisieren. Dabei wird eine neue Syntax eingeführt:

```
x(x < 0) = 0;  // Setze x auf 0 wo x negativ ist
```

Dieser Ausdruck beschreibt einen parallelen Vergleich und eine bedingte Zuweisung in den Einträgen wo der Vergleich `true` zurückliefert. Semantisch entspricht dieser Ausdruck der folgenden Schleife:

```
for (std::size_t i = 0; i < x.size(); ++i)
  if (x[i] < 0)
    x[i] = 0;
```

Die allgemeine Sprachregel ist:

vector-object (*mask-object*) *assignment-operator initializer-clause*.

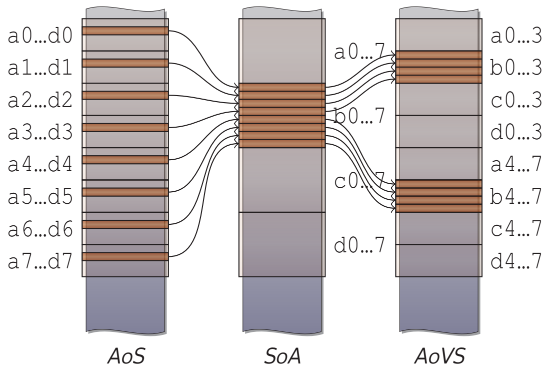
Es ermöglicht damit eine effiziente Schreibweise für die Vektorisierung von Algorithmen welche datenabhängige Unterscheidungen machen müssen.

HÖHERE ABSTRAKTIONEN

Aufbauend auf `Vector<T>` und `Mask<T>` werden die Klassen *templates* `SimdArray<T, N>` und `SimdMaskArray<T, N>` definieren. Während `Vector<T>` und `Mask<T>` mit einer zielrechnerspezifischen Vektorbreite ($\mathcal{W}_\mathtt{T}$) arbeiten, kann die Anzahl der Einträge bei `SimdArray` vom Entwickler gewählt werden. Die Implementierung nutzt dann die beste Zerlegung in Vektorregister und skalare Register für das gewählte `N`. Dieser Datentyp ist insbesondere von Bedeutung, wenn zwischen verschiedenen Vektordatentypen konvertiert werden muss, da $\mathcal{W}_\mathtt{T}$ und $\mathcal{W}_\mathtt{U}$ im allgemeinen nicht gleich sind. Außerdem lassen sich manche Algorithmen klarer und effizienter mit einer festen Vektorbreite beschreiben.

Der `simdize<T>` Ausdruck ermöglicht die generische Vektorisierung von Datenstrukturen. Dabei wird ein Argument `C<Ts...>` rekursiv übersetzt in `C<simdize<Ts>...>`. Ist `T` ein fundamentaler Datentyp ergibt `simdize<T>` entsprechend `Vector<T>`, `Mask<T>`, `SimdArray<T, N>`, oder `SimdMaskArray<T, N>`. Sind nun noch der Funktionsaufruf `get<N>(std::declval<T>())` und der konstante Ausdruck `std::tuple_size<T>::value` definiert, dann können die Funktionen `simdize_extract` und `simdize_insert` genutzt werden, um Objekte vom Typ `T` an einem gegebenen Index zu extrahieren oder einzufügen. Damit funktioniert `simdize<T>` also mit `std::tuple` und jedem anderen Klassen *template*, welches die `tuple` Schnittstelle implementiert.

Der `simdize<T>` Ausdruck unterstützt damit eine Aufteilung der Daten im Speicher, die weder *Array of Struct* noch *Struct of Array* (vergleiche [11]) folgt. *Array of Struct* ist die klassische skalare Herangehensweise, während die nicht typbasierten Ansätze zur expliziten Datenparallelisierung *Struct of Array* empfehlen. Mithilfe von `simdize<T>` kann man stattdessen leicht eine *Array of Vectorized Struct* Speicherorganisation erstellen. Dies ist cacheeffizienter und konzeptionell näher am objektorientierten Programmieren als *Struct of Array*. Die verschiedenen Speicherorganisationen sind in der folgenden Abbildung illustriert:



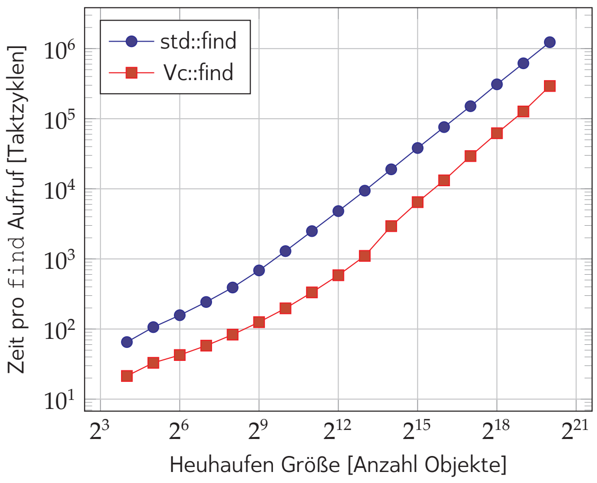
ANWENDUNGEN

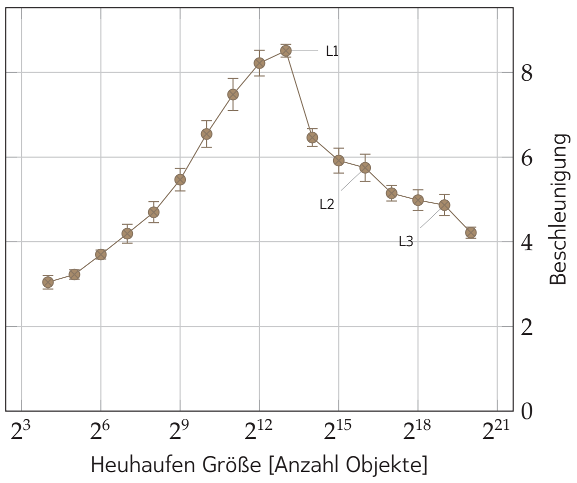
Die Vc Bibliothek ist eine Implementierung dieser Datentypen und als freie Software für jeden Nutzer frei verfügbar und wird inzwischen weltweit eingesetzt. Ich habe die Vc Bibliothek entwickelt, um datenparallele Datentypen zu erforschen und als eine Lösung für das explizit datenparallele Programmieren. Um die Relevanz der Bibliothek zu zeigen werde ich im Folgenden zwei Beispiele diskutieren.

SUCHE

Die Suche ist eine häufige Aufgabe in Computeranwendungen, welche in vielen verschiedenen Varianten vorkommt. Fast immer besteht bei einem Suchalgorithmus eine Abhängigkeit des Ausführungspfades von den Daten. Damit kann weder der Compiler noch der Algorithmus entscheiden wie viele Iterationen eine Schleife in der Suche brauchen wird bevor die Schleife startet. Dies ist für schleifenbasierte Vektorisierer ein Ausstiegskriterium. Die typbasierte Vektorisierung kann aber relativ einfach für Suchalgorithmen verwendet werden. Im Zweifelsfall werden ein paar Vergleiche zu viel ausgeführt, was aber das Ergebnis nicht beeinflusst.

Entsprechend kann der `std::find` Algorithmus [9, §25.2.5] mithilfe der Vc Bibliothek implementiert werden. Der Vergleich von `std::find` und “`Vc::find`” zeigt eine deutliche Effizienzsteigerung durch Vektorisierung:



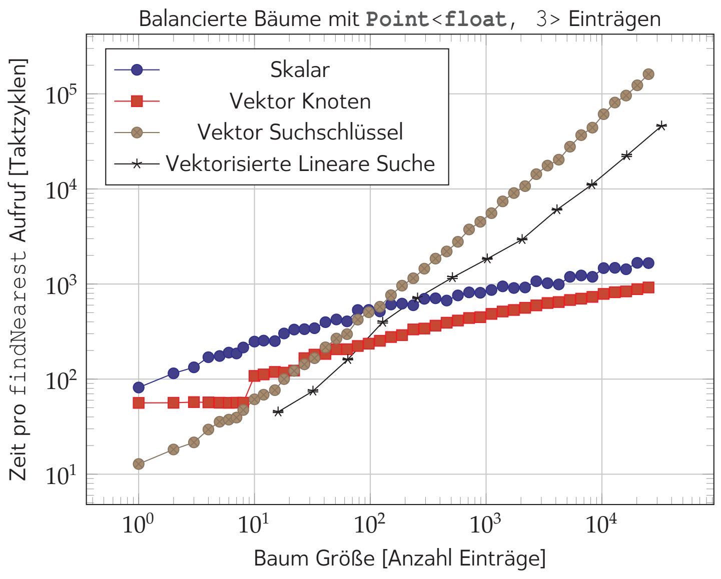


Dabei ist, dank des Einsatzes von **`simdize`**`<T>`, die vektorisierte Implementierung “`Vc::find`” immer noch eine generische Funktion.

Komplexere Suchalgorithmen werden meist nicht nur als alleinstehende Algorithmen definiert, sondern benötigen auch entsprechend zugeschnittene Datenstrukturen. So wird bei der Suche der nächsten Nachbarn häufig k-d Tree [2, 6] eingesetzt. Dieser Suchalgorithmus erwartet eine spezielle Baum Struktur mit einem (skalaren) Diskriminatorwert pro Knoten. Entsprechend kann diese Baum Struktur nicht direkt datenparallel durchlaufen werden.

Die k-d Tree Datenstruktur kann aber so verändert werden, dass in jedem Knoten $\mathcal{W}_T$ Diskriminatoren gespeichert werden. Die *INSERT* und *FINDNEAREST* Algorithmen müssen dafür entsprechend angepasst werden. Die Schnittstellen der Algorithmen bleiben dabei gleich zu der klassischen, skalaren k-d Tree Variante: *INSERT* fügt einen Eintrag in den Baum ein und *FINDNEAREST* sucht den einen Eintrag im Baum für den die Abstandsfunktion minimal ist.

Die folgende Abbildung zeigt einen Vergleich verschiedener Implementierungen für die Suche nächster Nachbarn:



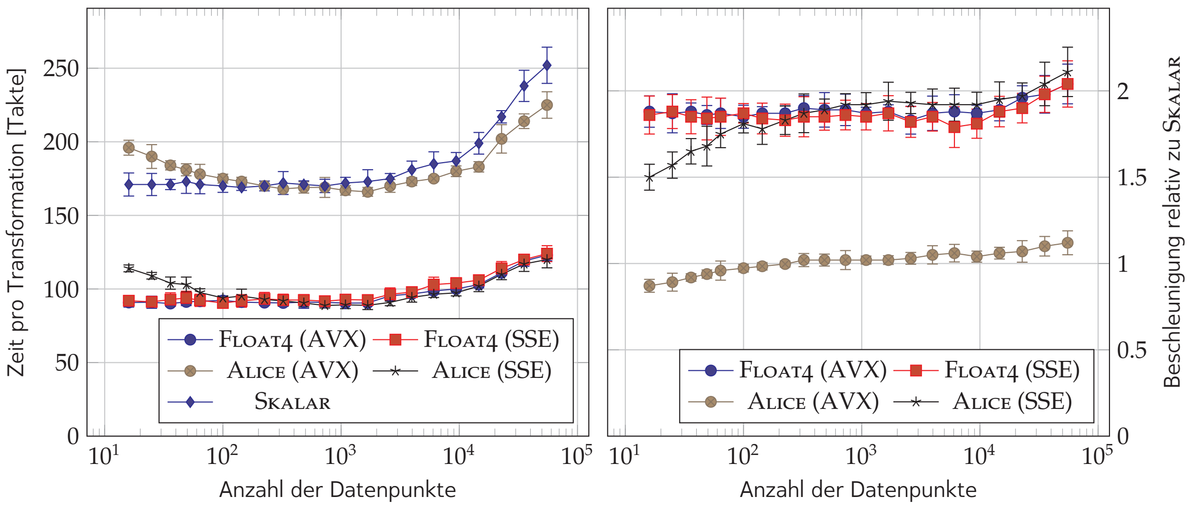
Die erste Kurve (—●— Skalar) zeigt eine klassische, skalare Implementierung des k-d Tree *FINDNEAREST* Algorithmus. Die zweite Kurve (—■— Vektor Knoten) zeigt die mithilfe von **`simdize`**`<T>` vektorisierte k-d Tree Implementierung mit $\mathcal{W}_T$ Einträgen pro Knoten im Baum.

Die dritte Kurve (—●— Vektor Suchschlüssel) zeigt einen alternativen Ansatz. Der Suchalgorithmus wurde vektorisiert indem er nach $\mathcal{W}_T$ nächsten Nachbarn parallel sucht. Horn u. a. [7] und Foley u. a. [5] beschreiben diese Parallelisierung für einen GPU Raytracer. Man sieht, dass diese Vektorisierung nur für sehr kleine Suchmengen effizient funktioniert. Anstatt eines Suchbaumes bietet sich dann schon die vollständige nächste Nachbarn Suche an, bei der in einer linearen Suche alle Einträge in der Suchmenge verglichen werden. Diese letzte Implementierung (mit einem skalaren Interface in der `findNearest` Funktion) zeigt die vierte Kurve (—∗— Vektorisierte Lineare Suche).

ALICE

Das ALICE Experiment am CERN setzt Vc für eine effiziente Koordinatentransformation ein. Dabei wird mit 15 elementaren Splinefunktionen ein zweidimensionaler Punkt in einen dreidimensionalen Punkt transformiert. Die Berechnung der 15 Splinefunktionen wird in der ALICE Implementierung mithilfe von Vc parallelisiert. Da dort noch eine Version von Vc eingesetzt wird, die `SimdArray<T, N>` nicht unterstützt, wird die Vektorisierung nur durchgeführt wenn $\mathcal{W}_{\texttt{float}} = 4$. Daher läuft die Software auf der eigentlich effizienteren Hardware mit AVX-Instruktionen langsamer als auf Hardware mit SSE-Instruktionen. Ich habe daher noch weitere Implementierungen der Koordinatentransformation erstellt, um zu zeigen wie viele Varianten bei einer typischen Aufgabe auftreten können und welche Probleme sich dabei jeweils ergeben. In jedem Fall ist diese Anwendung durch das Gesetz von Amdahl [1] in ihrer Effizienzsteigerung limitiert.

Die Laufzeiten und die Beschleunigung relativ zur skalaren Ausführung von der originalen ALICE Implementierung und einer Implementierung mit `SimdArray<float, 4>` (FLOAT4) zeigt die folgende Abbildung:



Dies zeigt, dass die Effizienz dank Vc um einen Faktor zwei gegenüber der skalaren Implementierung gesteigert werden kann. Für die Anwendung macht dies einen großen Unterschied, denn die Transformation muss für alle Detektormesswerte durchgeführt werden. Weniger Rechenzeit an dieser Stelle bedeutet weniger Ressourcenanforderung und Energieverbrauch, was sich letztlich in geringeren Anschaffungs- und Betriebskosten niederschlägt.

FAZIT

SIMD Hardware darf heutzutage von Entwicklern nicht mehr vernachlässigt werden. Um sie aber effizient nutzen zu können, muss der Entwickler Verständnis und Intuition für diese Form der datenparallelen Verarbeitung erlernen. Nur so kann ein Entwickler die nötigen Datenstrukturen und Algorithmen entwerfen. Die Vektor- und Maskentypen der Vc Bibliothek bieten genau diese Programmierschnittstelle, welche gleichzeitig dem Entwickler helfen kann die Funktionsweise der datenparallelen Ausführung zu erlernen. Dabei bilden die Klassen `Vector<T>` und `Mask<T>` die elementaren Bausteine, auf denen höhere Abstraktionen zum effizienteren Programmieren aufgebaut werden können.

Mit Vc vektorisierte Anwendungen zeigen, dass wertvolle Effizienzsteigerungen möglich sind. Gleichzeitig zeigt die nähere Untersuchung, dass diese Verbesserungen nicht durch eine automatische Schleifenvektorisierung möglich sind. Die Vektortypen ermöglichen folglich die Entwicklung eines portablen und verständlichen Quelltextes, welcher zu effizienteren Programmen führt als ohne Vektortypen möglich wäre.

LITERATUR

[1] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, S. 483–485. DOI: `10.1145/1465482.1465560`.

[2] Jon Louis Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Communications of the ACM* 18.9 (Sep. 1975), S. 509–517. ISSN: 0001-0782. DOI: `10.1145/361002.361007`.

[3] M. Flynn. "Some Computer Organizations and Their Effectiveness". In: *Computers, IEEE Transactions on* C-21.9 (1972), S. 948–960. ISSN: 0018-9340. DOI: `10.1109/TC.1972.5009071`.

[4] Agner Fog. *Instruction tables*. Manual. Copenhagen: Copenhagen University College of Engineering, 2012. URL: `http://www.agner.org/optimize/`.

[5] Tim Foley und Jeremy Sugerman. "KD-tree Acceleration Structures for a GPU Raytracer". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '05. New York, NY, USA: ACM, 2005, S. 15–22. ISBN: 1-59593-086-8. DOI: `10.1145/1071866.1071869`.

[6] Jerome H. Friedman, Jon Louis Bentley und Raphael Ari Finkel. "An Algorithm for Finding Best Matches in Logarithmic Expected Time". In: *ACM Trans. Math. Softw.* 3.3 (1977), S. 209–226. ISSN: 0098-3500. DOI: `10.1145/355744.355745`.

[7] Daniel Reiter Horn u. a. “Interactive k-d Tree GPU Raytracing”. In: *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. ACM, 2007, S. 167–174. DOI: 10.1145/1230100.1230129.

[8] *Intel ® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. Juni 2013. URL: http://www.intel.com/products/processor/manuals/.

[9] *ISO/IEC 14882:2011. Information technology — Programming languages — C++*. Standard. ISO/IEC JTC 1/SC 22, 1. Sep. 2011. URL: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=50372.

[10] *Software Optimization Guide for AMD Family 10h and 12h Processors*. Publication # 40546. Advanced Micro Devices. Dez. 2010. 346 S. URL: http://developer.amd.com/documentation/guides/Pages/default.aspx (besucht am 25. 01. 2011).

[11] Robert Strzodka. “GPU Computing Gems Jade Edition”. In: *GPU Computing Gems Jade Edition*. Elsevier, 2012, S. 429–441. ISBN: 9780123859631. DOI: 10.1016/B978-0-12-385963-1.00031-9.