Vc: A C++ library for explicit vectorization

Matthias Kretz* and Volker Lindenstruth

Frankfurt Institute for Advanced Studies & Institute for Computer Science, Goethe University Frankfurt

SUMMARY

It is an established trend that CPU development employs Moore’s Law to improve in parallelism much more than in scalar execution speed. This results in MIMD and SIMD enhancements, of which the MIMD developments have received the focus of library research and development in recent years. To make use of the latest hardware improvements, SIMD must receive a stronger focus of API research and development since the computational power can no longer be neglected and often auto-vectorizing compilers cannot generate the necessary SIMD code, as will be shown in this paper. Nowadays, the SIMD capabilities are sufficiently significant to warrant vectorization of algorithms requiring more conditional execution than was originally expected for SSE to handle. The Vc library (http://compeng.uni-frankfurt.de/?vc) was designed to support developers in the creation of portable vectorized code. Its capabilities and performance have been thoroughly tested. Vc provides portability of the source code, allowing full utilization of the hardware’s SIMD capabilities, without introducing any overhead. Copyright © 2011 John Wiley & Sons, Ltd.

KEY WORDS: SIMD; C++; data-parallel; AVX; LRBni; SSE; optimization; Vc; vectorization

This is the pre-peer reviewed version of the following article:

1. INTRODUCTION

The time where SSE‡ was only useful for specialized tasks (such as in multimedia) has come to an end. Operations on 128-bit registers are already capable of executing as fast as the equivalent scalar operations since a few years. Thus, a speedup of factor four is possible if the data can be provided to the vector registers fast enough.

In general, applications can increase both calculation throughput (increased parallelization) and I/O throughput (vector loads and stores, which possibly employ cache optimizations) when using SSE. But two fundamental problems remain:

1. SSE (and SIMD in general) is difficult to program for. Compilers generally provide three options: auto-vectorization, linking assembler routines, or using intrinsics.
2. Vectorization of a given algorithm requires a design that can make use of the data parallelism in the application. The data structures must then be optimized for fast vector loads and stores. These two considerations influence each other.

The second point implies that auto-vectorization can not magically yield good results. If the designer/programmer does not take vectorization into account, the result of auto-vectorization will either depend on chance or not work at all. The relatively simple inner loop of the Mandelbrot
algorithm shows that auto-vectorization does not work, while explicit vectorization yields a speedup of the vector width (see Sections 4.3 and 5.3 for details).

New vector instructions will be available in server hardware soon:

- The Intel Sandy Bridge CPU architecture includes AVX (Advanced Vector Extensions). AVX doubles the width of the SIMD registers to 256 bits and provides a large set of single- and double-precision floating-point instructions\(^1\). All new instructions and the old SSE instructions are available as ternary operations, thus eliminating the need for many of the currently required mov instructions. AMD will introduce AVX with their upcoming Bulldozer CPU architecture, too.
- The (canceled) Intel Larrabee GPU and the announced Intel Many Integrated Core (MIC) architecture introduced a 512-bit wide vector unit with much improved support for general purpose code compared to SSE or even AVX.

These new developments show that the failure to employ SIMD in computationally intensive applications will become more severe with regard to the unused computation power.

1.1. Application of SIMD in Track Reconstruction

As a real life application we used the data produced by one of the experiments (ALICE [1]) of the Large Hadron Collider accelerator at the European high energy physics research institution CERN. This experiment produces more than 20 Terabytes/sec and requires to reconstruct the trajectories of thousands of particles produced in collisions with kHz rates. Each particle trajectory is composed of hundreds of detector hits. This reconstruction of trajectories from the underlying hits is based on cellular automata and Kalman filter. This is a very computational intense effort.

We conducted a study to investigate the use of the Intel Larrabee GPU in this reconstruction of trajectories (track-finding) [2]. This motivated the development of an abstraction for SIMD [3]. Since the Larrabee would only operate at \(\frac{1}{32}\)th of its capabilities without the use of SIMD instructions, it was a requirement to port the track finder to LRBni\(^1\). To do so with LRBni intrinsics or even Assembler would have made the code unportable and dependent on this one specific GPU.

The track-finding software employs a highly parallel algorithm, which at first appears to be impossible to vectorize, due to high dependency of the instruction stream on the data (Class 2 problem; see Section 2.1). But a horizontal vectorization over the different measurement points or tracks is possible. It requires reorganization of the data (with different order depending on the vector unit), the use of gathers and scatter, and masked vector execution. Auto-Vectorization of the tracker by a compiler was therefore not possible. A compiler is not allowed to do the required data reorganization according to the specifications of C/C++ and will not vectorize code that contains (many) branches.

Employing the Vc library, as presented here, enabled the tracker code to compile for scalar instructions, SSE2 up to SSE4, and LRBni.

1.2. Previous Work on SIMD Abstractions

The EVE [4] 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, but it has never been done. The library has never been published with a Free Software license. Also, the main emphasis is put on vectorization of calculations on larger data structures with more or less homogeneous treatment. This made it unsuitable for the requirements of the track finder, which requires heterogeneous data processing and a maintained code base that follows current hardware improvements.

Another library wrapping AltiVec exists, where MMX and SSE up to SSE3 were added later on, called macstl [5]. This library focuses on a main class for processing which is an improved STL 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, either.

\(^1\)AVX does not include 256-bit instructions for integers. This is a major shortcoming for porting from SSE and makes the development of algorithms requiring both floating point and integer logic more complicated.

\(^1\)LRBni is short for Larrabee new instructions.
The Kalman filter, the most compute-intensive part of the track finder, has previously been vectorized, which was presented in [6]. 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 [7].

1.3. Limits of Compiler Auto-Vectorization

The most prominent work on SIMD abstraction is done on compilers. All major C/C++ compilers can auto-vectorize loops. But this is not the solution for all problems. The compiler’s success depends on specific loop layouts and the data structures. Additionally, the compiler needs to determine the dependencies between data accesses, where the developer often has more information than is expressed in the programming language. Thus, “only certain types of loops can be automatically vectorized, and in most cases user interaction with the compiler is needed to fully enable this.” [8]

Even if a given code can be auto-vectorized, this poses a very special maintenance burden. Any changes to the code could introduce a reason for the auto-vectorization to fail, and thus lead to significant slowdowns. Since the effect of a given change on the auto-vectorizer is not obvious, it increases the complexity of maintenance and will keep vectorization efforts an afterthought.

2. VECTORIZATION APPROACHES

Every application contains data parallelism. But not every data parallelism can easily be utilized for vectorization. It is helpful to classify the parallelism in the application that should be vectorized. This classification can help to understand the possible approaches and how to break down the problem best:

2.1. Classification

Class 1 data parallel, where the processing of multiple data sets is trivially possible with a single instruction stream
Vectorization is possible and easily accelerates the code

Class 2 data parallel, but with a dependency of the instruction stream on the data (branching)
Vectorization is possible, but traditionally not done

Class 3 not data parallel; thus every data set requires a separate instruction stream
Vectorization is useless; Multithreading can be used

Problems of Class 1 can often be sped up when vectorized, unless the memory throughput is the only limiting factor. Note that read and write accesses to the cache can be accelerated using SIMD, thus the real limit is the throughput to/from the RAM. (See Section 5.1.3 for details.) Examples for Class 1 problems are dense linear algebra and many image processing tasks. Another example is depicted in Figure 1.

Problems of Class 2 require a more thorough understanding of the parallelism and conditionals involved. The gain of vectorization seldom reaches the speedup which is possible for Class 1 problems. The dependencies on the data require execution of several branches of the code for one SIMD register, thus often increasing the number of instructions that have to be executed per value considerably. Figure 2 shows an example of a Class 2 problem that will be discussed in more detail later. Sometimes problems of Class 2 can be transformed into Class 1 by reorganization of the data or a change of the vectorization direction, as explained in the following.

2.2. Vectorization Direction

Most given problems contain data parallelism that can be utilized for vectorization. They can be divided according to:

**Vertical** Per object to be processed (in the example in Figure 3, a 3D coordinate \(x, y, z\) constitutes an object), often many of the calculations can be executed in parallel. Instruction-level parallelism makes use of this fact to execute several instructions at the same time. If this parallelism is used for vectorization, the amount of parallelism left for instruction-level parallelism might get reduced, as is the case in the example of the scalar product.
for(int i = 0; i < FrameSize; ++i) {
    si[i] = static_cast<signed short>(
        min(65535.f, max(0.f, 
            (si[i] + 1.f) * 32767.5f)) 
        - 32768);
}

Figure 1. Example for a Class 1 problem: Audio samples that were processed as floating point values need to be converted to 16-bit signed integers to be written to the audio device. This task can easily be executed four times faster on x86 CPUs by employing SSE. (It can be further improved by employing the packsswb instruction of SSE.)

int n;
std::complex<float> z = c;
for (n = 0; n < maxIt && norm(z) < S; ++n)
    z = P(z, c);

Figure 2. Simple example for a Class 2 problem: The inner loop of the Mandelbrot algorithm needs to determine the number of iterations required until ||z|| ≥ 2. While each evaluation of P is a Class 1 problem, the condition when to exit the loop and how to count n makes the inner loop a Class 2 problem.

**Horizontal** Several objects, which have to be processed in the same way, can be combined into vectors. Instead of one register containing one value of one object, the code is modified to use SIMD registers containing one value of several objects. In the example in Figure 3 four scalar products from four objects are executed in parallel, still fully employing the instruction-level parallelism. This is possible because of the data parallelism inside the scalar product.

The two different directions have different scaling behavior. Vertical vectorization scales well with larger objects. Larger vector registers on the other hand require a minimum object size to be efficient (a 3D vector with x, y, z stored in a vector register of 16 values makes use of only 19% of the hardware). Horizontal vectorization scales well with larger numbers of objects, while the object size itself should not be too large to keep the working set at a size that can fit into the cache (or even better, just a few registers). Larger vector registers thus require a minimum number of objects to be processed to be efficient.

It is not always possible to determine a single best vectorization direction for a given application. Sometimes it is better if the outer vectorization direction is horizontal, while some inner loops should better use vertical vectorization instead. This switch of the vectorization direction is easily possible by iteration over the entries of the horizontal vectors. It has shown its applicability in the vectorization of the tracker described in Section 1.1 [3].
3. VC: ABSTRACTION FOR VECTORIZATION

The C and C++ languages have built-in types for most types the x86 architecture supports in hardware except for the MMX, SSE, and AVX registers. These types are accessible in an extension to C/C++ by the use of intrinsics. This extension defines the types __m64 and __m128 (since SSE2 also __m128d and __m128i; with AVX it supports __m256, __m256d, and __m256i).

The new types introduced with the intrinsics have the shortcoming that they are defined for one hard-coded vector-register width only. The C/C++ int type, for example, need not be implemented as a certain register width, but is only specified as a register width reflecting the natural size of the host machine [9]. To achieve the same for a vector type, it would have to be translated to the correct register width (number of entries in the vector) at compile time, depending on the target architecture. Types of this kind are introduced by the Vc library:

float_v Native vector register which can hold 32-bit floating point values. With SSE, this uses the __m128 type internally. With a different underlying vector architecture this would use a different type (such as __m256 for AVX).

double_v Native vector register holding 64-bit floating point values.

int_v and uint_v Native vector register\(^\d\) with 32-bit signed or unsigned integers.

short_v and ushort_v Native vector register with 16-bit signed or unsigned integers.

The types behave very much like the related scalar types and often can simply be substituted. Thus, it is possible to replace `std::complex<float>` by

---

\(^\d\)On x86 there will be three native integer vector registers: MMX (64 bit), SSE (128 bit), AVX (256 bit). While the MMX registers are obsolete with SSE, the AVX instructions do not support full 256-bit integer operations.
// infix notation:
float_v c = a + b;
// prefix notation:
__m128 c = _mm_add_ps(a, b);

Figure 4. Code examples for infix and prefix notation. The upper example uses Vc whereas the lower example uses SSE intrinsics.

std::complex<float_v>. Additionally, there are useful class and member functions as well as defined constants making the Vc types very flexible low-level types, but with a high-level API.

3.1. API

The application programming interface of Vc is the most important feature allowing efficient development of vectorized algorithms. The software developer’s duty is increasingly becoming more and more complex, especially with the latest requirement to develop and implement parallel algorithms. The complexity of understanding the problem and designing a good parallel algorithm can be lowered by acquiring knowledge of supporting methods and concepts (see Section 2). The complexity of implementing a vectorized algorithm can be lowered considerably compared to the readily available options of developing in assembler or employing intrinsics. The ultimate goal is to provide an API for vectorization that allows developers to freely express themselves in a way that creates readable code, thus helping the developer to focus on the design and lowering the entry barrier for other developers. The presented API has been used successfully in entry-level classes at the Goethe University, Frankfurt to program highly efficient vectorized algorithms.

The C++ language allows the creation of a set of vector types that can be used and understood very easily because of their similarity to the built-in types. This is made possible through the implementation of overloaded arithmetic, binary, logical, and comparison operators of the vector classes. It allows the application developer to use infix notation, a more intuitive and readable syntax compared to the prefix notation with additional namespace overhead provided by the C-intrinsics (see Figure 4).

Since many different vector architectures exist and the chipmakers’ roadmaps tell us that their number will continue to increase, the differences should be hidden behind an abstraction. This is in line with the idea of the C and C++ languages, which provide a low-level programming language but with portability to many different target systems. First and foremost, this requires an abstraction of the width of the vector registers. Thus, every vector type of Vc uses a vector width that depends on the target system. While this makes writing portable application code slightly more difficult, it allows the same low-level access to vector types as does C/C++ for scalar types. To allow development of portable code, the API provides a simple enum member which is defined as the number of elements in one vector object.

Another important challenge is the requirement of writing conditional code. Traditionally, every condition in the algorithm has been implemented using if statements or the ternary operator. Developing vectorized code requires the use of conditional assignment/write-masking. This is not supported in the original C/C++ syntax. Therefore, a new syntax was invented. Every assignment to a vector object can be prefixed by a mask in parenthesis. This will then be translated to the best way of doing conditional assignment on the respective target system. Generally, the assignment syntax looks like this:

<vector object> [(<mask object>)] <assignment operator> <vector object>.

Developers can easily get accustomed to it because the code, such as v1(v1 < 3.2f) *= 1.1f, reads naturally as: all entries of v1 where v1 < 3.2f shall be multiplied by 1.1f.

---

†C does not allow function overloading. Therefore, there must be a different function name for each arithmetic function for every data-type.
‡In C/C++ terminology this includes all the +, -, *... operators, too.
typedef EntryType AliasingEntryType __attribute__((may_alias));

AliasingEntryType &m(int index) {
    return reinterpret_cast<AliasingEntryType *>(&data)[index];
}

EntryType m(int index) const {
    return reinterpret_cast<const AliasingEntryType *>(&data)[index];
}

Figure 5. Class used for the basic storage and access. Note that the may_alias attribute is only used for the non-const scalar access function.

The Vc API allows calling a subset† of the familiar math functions from <cmath>. Currently the sin, cos, tan, asin, atan, and atan2 functions are supported. For SSE, they are implemented in a similar manner as in the Cephes library [10].

C++ has a stricter type system than C, but it still allows for many automatic type conversions. The typically used conversions are automatic promotion of types from smaller types to larger types. Automatic demotion of types normally results in a compiler warning. The Vc vector types implement a similar behavior, allowing automatic type conversions where they can be safely done (promotion of types at no loss of information). Since it is not possible to provide type conversions that produce a compiler warning with standard C++, any kind of unsafe automatic conversion is avoided, producing compiler errors when attempted. Thus, the Vc types are slightly stricter, requiring more explicit casting than the built-in types. One of the most important automatic type conversions is the conversion from scalar values to vectors. This results in a broadcast of the value to all entries of the vector and thus produces the intuitively correct result for code such as \( v + 3.f \): adding the value 3 to all entries of \( v \).

The correct use of simple arrays for fast vector access is a non-trivial task. Vc provides the class Memory to ease the use of arrays with vectors. It automatically aligns and pads the memory to allow fast vector access in the full index range. Additionally, it provides member functions to easily iterate over all vectors or scalars in the array.

3.2. Implementation

Fully documenting the implementation of Vc would go beyond the scope of this article. Therefore, two major features will be highlighted in the following.

3.2.1. Zero-Overhead  It is essential that the use of Vc does not introduce additional overhead. The benchmarks in Section 5.1 and especially Figure 9 show that the abstraction on top of intrinsics does not introduce any overhead.

Vc achieves this by providing almost all of its code as inline functions. This is the most obvious step, which is required to allow compilers to eliminate the Vc classes. But further consideration is required on how the actual vector data is stored and how the different vector widths and instructions can be supported.

The Vc classes use __m128 (or __m256 or __m512) for storage. Vc implements scalar access to the vectors using reinterpret_casts instead of unions. The __m128 must first be cast to a type that is defined with the may_alias attribute‡. This attribute tells the compiler that it should not apply the strict aliasing rules defined by the C/C++ standards. In principle this results in the same behavior as if a union is used. The important difference is that the union inhibits aliasing optimization everywhere while the explicit cast to an aliasing type inhibits aliasing optimization only in the places where it is required. Details about the performance can be found in Section 5.1.2.

Figure 5 shows the code required to implement the scalar access with GCC. This low level access is abstracted into a minimal class, easily allowing substitution with a union-based storage or other aliasing implementations, should any other compiler require different treatment.

†The full set can be supported. It was not done for lack of requirement and time.
‡The attribute syntax is a GNU extension supported by GCC and all GCC compatible compilers.
Reaching highest performance while targeting different vector architectures requires different implementations of a common API. Consider the addition operator as a simple example: The name of the intrinsic function uses the _mm_prefix for SSE, _mm256_ for AVX and _mm512_ for LRBNi. Thus, Vc requires different code for every target architecture it supports.

To ease maintenance, research, and development, the number of #ifdefs is kept to a minimum. Therefore, the three main implementations Scalar, SSE, and LRBNi all define their functionality in different classes (i.e., Vc::SSE::Vector<float> vs. Vc::LRBNi::Vector<float>). This easily allows different vector widths in the different Vc implementations. It is required that a common and portable API is provided. Thus, any function that depends on a certain vector width cannot be part of the public API. An example of such a function would be the set-constructor, which for Scalar::Vector<float> would only take one argument while it would take 4 arguments for SSE::Vector<float> and 16 arguments for LRBNi::Vector<float>.

Vc only provides public functions that are independent of the vector width and provides higher level functions where lower level functionality cannot be provided in a vector width independent way. An example is the deinterleave function which fills two vector objects from a given pointer to memory such that the first vector holds the values from memory with even indexes and the second vector the values with odd indexes. For SSE, this is an abstraction of the _mm_unpacklo and _mm_unpackhi intrinsics. Depending on the number of values in the vector, a different number of unpacks is required. Letting application code implement the deinterleaving of memory would lead to unportable code, because the code would depend on the vector architecture.

From this library design it follows that only source compatibility between the different implementations can be supported. Imagine a project that employs Vc and builds a library and an executable which links to the library. If the library is compiled with the scalar implementation of Vc and the executable is compiled with the SSE implementation then any interface between the library and executable that passes a Vc datatype will break at link time.

3.2.2. Conditional Assignment

Masking vector operations build the basis upon which non-trivially data-parallel algorithms can be vectorized. Thus, it requires both an intuitive and powerful API and the most efficient implementations, which take as much context into consideration as possible.

The main pattern used by Vc to implement masked operations is a C++ class called MaskedVector. This class has two data members: a reference to a SIMD vector and a Mask object. It implements all relevant assignment operators such that the referenced vector is only changed in those entries where the mask is set. Objects of class MaskedVector are created by operator() (Mask) of the main vector class. This operator simply returns a new object of MaskedVector.

This pattern can be extended to include more context, which is especially useful for fused multiply-add and LRBNi. All binary operators, except for assignment operators, are implemented to only return an object with references to the operands and the operation encoded in their type. Only when an object of this kind is passed to an assignment operator will the actual operation be executed. This allows further optimizations to be implemented in the assignment operator which now can execute different code paths depending on the previous operations. This is a well documented pattern called Expression Templates [11] and has shown its strengths already in template libraries such as Eigen [12]. For LRBNi, this pattern allows execution of all operations leading to a write-masked assignment operator with write-masking, which can save power and thus increase the thermal headroom.

4. APPLICATION OF VC

4.1. Benchmarks

It is important to verify every aspect of performance of the Vc library to ensure that no overhead is introduced by Vc. Benchmarks of the primitives and of higher level abstractions can help to identify
VC: A C++ LIBRARY FOR EXPLICIT VECTORIZATION

__m128 tmp = _mm_add_ps(a, b);
asm volatile("" : : "x"(tmp));

Figure 6. Trick to inhibit dead store elimination

problems in Vc before they impact the performance of end-user applications. Obviously, many of the functions in Vc will be called very often, thus impacting applications already if a single unnecessary instruction is executed. Additionally, the benchmarks help to understand the possibilities offered by vectorization and how different hardware and compilers affect the results.

The benchmarks in Vc use the time stamp counter (TSC) of the CPU to measure the number of cycles between a start- and an end-point. † The code in between is always written in a way that it runs at least a few million cycles. This minimizes the effect of the large variance and overhead of the TSC instruction.

Modern compilers excel at finding dead code and removing it from the final executable. A naïve approach to repeatedly calling a single operation is to generate one input datum and then run the operation many times on that same datum. The compiler will notice that the input data is constant and induce that the output data of the operation is constant. After common subexpression elimination, the optimized code will thus call the operation that should be benchmarked only once, keep the result, and store the result repeatedly.

Storing the result is another pitfall of writing a benchmark: If the compiler can prove that the result will not be used/needed, then all code that produces this result and has no other side effects is identified as dead code and eliminated. A simple solution is to use global variables to store the results, as the compiler cannot make any assumptions on how that data is used or not. (Other code, added through a later linking stage or even an LD_PRELOAD, could make use of that value.) The Vc benchmarks use a less elegant solution, which does not introduce any overhead. An inline assembly call, which does not contain any executable code, instructs the compiler that a given variable is required in a register at the specified point in the code (Figure 6). This eliminates the overhead of the store to the global variable.

The time measurements require repetition of the benchmarked operation many thousands (or millions) of times. Additionally, it is important to benchmark with a broad range of input values, in case the values have an influence on the performance. This requires reading of the input data from the CPU caches. It is best if the input data size does not exceed the size of the L1 cache if the test in question should show the performance of a calculation. To still be able to repeat the operation often enough, the code can loop over the input data multiple times as the compiler does not have the freedom to keep the results around for subsequent loops.

The operating system schedules processes as it determines is best. For optimal benchmark results, the OS needs to be told to not preempt the running benchmark. This is achieved by executing the benchmark with real-time priority.

To discuss instruction performance, there are two important numbers:

- The latency specifies the number of cycles the instruction needs to deliver the result so that it can be used by subsequent instructions.
- The throughput specifies the number of cycles that have to elapse before the same instruction can be issued again (as long as the input data is available, of course).

For example, the SSE float addition instruction adps has a latency of three cycles and a throughput of one cycle on Intel CPUs. Therefore, if three adps instructions are executed in succession, the first result is ready after three cycles, the second result after four cycles, and the third result after five cycles.‡

The following four benchmarks are presented and discussed in this article:

**Peak FLOP** This benchmark implements a loop of (vector) multiplications and additions in such a way that eight independent variables/data streams (each may consist of a whole vector

---

† Note that recent CPUs implement a stable TSC (the TSC frequency is independent of the clock) and thus different P-states (Clock frequency scaling from power-management or turbo mode) of the CPU will not be normalized automatically.

‡ Of course, this is only true if there is no data dependency.
of values) are processed. No memory access is required. The number of independent data streams is chosen according to the combined latency over throughput ratio of the multiply and add instructions, which is 4/1 + 3/1 on recent Intel CPUs and 4/1 + 4/1 on recent AMD CPUs. The code is written to make use of three levels of parallelism: SIMD, pipelining, and superscalar execution. There are four independent implementations of the same calculations. The first is written in Assembler, the second using Vc’s float_v, and the third using SSE intrinsics. The fourth implementation is a slight modification of the Vc code. It uses 32 independent float variables instead of the 8 float_v variables and is compiled explicitly either with or without auto-vectorization.

**Union vs. Casts** It was mentioned in Section 3.2.1 that casts are used to alias memory between vector types and scalar types. To show the importance of this matter, the core storage class was reimplemented with a union and the Peak FLOP benchmark was used to determine the difference.

**Memory I/O** This benchmark determines the throughput of loads and/or stores for different datatypes and different array sizes. The array sizes are determined from the cache sizes the CPU reports. The test program simply loops over the array multiple times and reads from and/or writes to it. No prefetch instructions are used.

**Arithmetics** The four basic arithmetic operators are the building blocks of almost all other mathematical operations (there are a few exceptions such as square root, look-up tables of the LRBni math functions and the FPU math functions), so the performance of these operations is what matters most for all calculations. To compare the performance of the arithmetics of different vector instructions, a simple loop calls the operation repeatedly on random input data. The benchmark does not measure the actual latency or throughput of the instructions as there is overhead from the loop and loads involved.† (This is not necessarily a bad thing, as thus the benchmark relates slightly more to real world applications where data is not always readily available in registers.)

### 4.2. Unit Tests

Unit tests are tests that ensure a specific behavior and result from small units of a library or program. In the case of Vc, unit tests are used to ensure API/source compatibility between the different implementations (Scalar, SSE, and LRBni) and correct results from all the operations. Since SSE support via intrinsics is not as well tested in compilers as the standard scalar types, it is important to verify for as many combinations of compilers (GCC, ICC), compiler versions (GCC 4.1 – 4.5), and target architectures (Intel/AMD, 64-/32-bit) as possible that the Vc library works as expected.

### 4.3. Mandelbrot

The calculation of the Mandelbrot set is a simple example that shows the power of vectorization and the Vc API. To calculate the Mandelbrot set, a function \( P(z, c) = z^2 + c \) is repeatedly evaluated: \( z_{n+1} = P(z_n, c) \) (with \( z, c \in \mathbb{C} \)). The value of \( z_0 \) is 0; and \( c \) is taken as the \( x \) and \( y \) coordinates on the image (\( c = x + iy \)). If the \( z_n \) diverge (which is simplified to \( |z_n| > 2 \)), the value is not in the Mandelbrot set and typically colored by how many iterations it required to reach the cut-off value. The remaining pixels are colored black, showing the famous Mandelbrot set.

A simple scalar implementation can be written as shown in Figure 7. A vectorization of this algorithm can be implemented relatively easily using the Vc float_v and int_v types. Figure 8 shows that the two inner iterations have been modified to process float_v::Size (= 4 when SSE is used) many values in parallel. Using SSE, the iteration variable \( x \) starts from \( [0, 1, 2, 3] \), adding 4 to all four entries on every loop. Then, the \( c \) vector is initialized and the innermost iteration over the \( P \) function is executed.

The number of iterations for all four entries is counted in one int_v vector. It is required that only selected parts of the vector can be incremented because if one of the values in \( \Re(z)^2 + \Im(z)^2 \) reaches \( S \) (defined as \( 4 = 2^2 \)) the iteration counter may not be incremented further. This is achieved

†This overhead can be eliminated, but then it would just measure the instruction throughput, which is already tested in the Peak FLOP benchmark.
typedef std::complex<float> Z;
for (int y = 0; y < imageHeight; ++y) {
    const float c_imag = y0 + y * scale;
    for (int x = 0; x < imageWidth; ++x) {
        Z c(x0 + x * scale, c_imag);
        Z z = c;
        int n = 0;
        for (; n < maxIt && fastNorm(z) < S; ++n) {
            z = z * z + c;
        }
        const uchar colorValue = (maxIt - n) * 255 / maxIt;
        colorizeNextPixel(colorValue);
    }
}

Figure 7. Scalar implementation of the Mandelbrot iteration

typedef std::complex<float_v> Z;
for (int y = 0; y < imageHeight; ++y) {
    const float_v c_imag = y0 + y * scale;
    for (int_v x = int_v::IndexesFromZero();
        !(x < imageWidth).isEmpty();
        x += float_v::Size) {
        Z c(x0 + static_cast<float_v>(x) * scale, c_imag);
        Z z = c;
        int_v n = int_v::Zero();
        int_m inside = fastNorm(z) < S;
        while (!(inside && n < maxIt).isEmpty()) {
            z = z * z + c;
            ++n(inside);
            inside = fastNorm(z) < S;
        }
        const int_v colorValue = (maxIt - n) * 255 / maxIt;
        int maxJ = min(int_v::Size, imageWidth - x[0]);
        for (int j = 0; j < maxJ; ++j) {
            colorizeNextPixel(colorValue[j]);
        }
    }
}

Figure 8. Vc implementation of the Mandelbrot iteration

using the mask object of type int_m. It can be understood as a vector of Booleans. The syntax ++n(mask) tells the compiler to create a masked increment operation.

Finally, the color value is calculated using an int_v vector, and then each scalar entry of the vector is read to assign the color values to the image. Note that the last iteration in Figure 8 is safe even for images with a width that is not a multiple of the vector size.

4.4. Test Hardware

The benchmarks and tests were executed on mainly two Intel and AMD systems:

Intel Nehalem  Xeon E5520; a Nehalem microarchitecture CPU

AMD Magny-Cours  Opteron 6172; a Family 10h microarchitecture CPU

An AMD K8 (Opteron 2218 HE) system was additionally used for the Peak FLOP benchmark.
5. RESULTS

5.1. Benchmarks

5.1.1. Peak FLOP Benchmark The results of the Peak FLOP benchmark, as described in 4.1, are shown in Figure 9. The Intel Nehalem CPU uses turbo mode (~10% increased clock) to reach 8.8 / 2.2 FLOP/cycle. Without turbo mode it shows the same performance as the Core 2 and AMD Family 10h CPUs. The AMD K8 is a previous generation microarchitecture which executes SSE instructions with 64 bits width only; therefore it executes only half as fast as current microarchitectures from AMD and Intel. Current microarchitectures provide single-cycle throughput and therefore reach eight FLOP per cycle.

The presented data results from a binary compiled with GCC. When the benchmark is built with ICC the assembly code shows the same performance, as is expected. But ICC compiles the Vc classes to inefficient code: In addition to the required eight mulps and addps instructions ICC adds 17 movaps and 32 64-bit mov instructions into the loop. Thus, the benchmark only reaches 1.73 ± 0.02 FLOP/cycle on a Nehalem CPU. The intrinsics code is almost optimized correctly (the loop contains two superfluous movaps instructions, which it does not contain in the GCC binary) and results in 5.10 ± 0.02 FLOP/cycle on the same CPU.

The results from the modified scalar code for optimal auto-vectorization are shown on the right-hand side in Figure 9. ICC is able to emit the optimal code from auto-vectorization, while GCC adds too many loads and stores into the loop. The same code which delivers good results with auto-vectorization delivers much worse results compared to the original scalar code when auto-vectorization is disabled. Without auto-vectorization the large number of independent data streams results in the need to use the stack for the data; the number of registers is no longer sufficient.

5.1.2. Union vs. Casts When a union of _m128 and an array of the scalar type is used instead of reinterpret_casts (as described in Section 3.2.1), a significant slowdown results when compiled with GCC. The Peak FLOP benchmark runs ca. 35 times slower when using a union. The slowdown is due to GCC adding many unnecessary loads and stores into the loop.

Using reinterpret_casts works optimally with GCC. The compiler emits the minimal number of loads and stores to implement code with mixed scalar and vector access. This is also visible in the 32-bit integer division benchmark (Figure 14), because SSE does not provide integer division instructions and thus Vc implements integer division with scalar instructions.

5.1.3. Memory I/O Figure 10 shows the performance of loads and stores on an Intel Nehalem CPU; Figure 11 shows the same benchmark on an AMD Magny-Cours. It can be seen that the Intel CPU can execute up to one load per cycle, irrespective of the size of the load. The AMD CPU on the other
hand can execute 1.7 SSE loads and 1.3 non-SSE loads per cycle on average (the quoted performance figures from AMD specify that two SSE loads per cycle are possible while the throughput of non-SSE loads is not documented [13]). Thus, the performance of scalar loads depends on the size of the datatype, while SSE loads can reach high performance, regardless of the datatype of the vector entries. Stores show a similar behavior, with the notable exception of double and SSE stores by the AMD CPU: two 64-bit stores can be combined to almost one 128-bit store per cycle, while full 128-bit stores perform slightly worse.

As is expected, the performance degrades with a larger array size. But still the SSE load/store instructions show better throughput than scalar loads/stores (Figure 12 and Figure 13).

When unaligned load and store instructions are used on aligned addresses the Intel and AMD systems show comparable performance to aligned loads and stores. If the addresses are unaligned, then the load/store throughput decreases.†

5.1.4. Arithmetics Figure 14 shows the results of the arithmetics benchmark. The measured execution time of each operation is normalized to the number of values it processes and then the quotient of the SSE operations vs. scalar operations is plotted. Thus, the plot shows the speedup that is possible if the vectors are fully used.

- The double and float operations all show the expected speedup of factor two or four, which corresponds exactly to the vector size.
- 32-bit integer division for SSE is implemented as four successive scalar divisions because no SSE integer division instruction exists. A speedup factor of 1 thus proves that the scalar fallback does not introduce unnecessary overhead.
- 16-bit integer division for SSE is not supported in hardware and therefore is implemented as two single precision float vector divisions in Vc, which is why it shows a speedup. The speedup is significant on the AMD CPU because the integer division performs relatively poorly.

†Quantification of unaligned loads and stores requires a lot of data. The benchmark ships with the Vc source package and can be used to acquire all necessary data.
• 32-bit integer multiplication is not supported with a dedicated instruction until SSE 4.1. This is why the AMD CPU (SSE4a) uses a sequence of multiplications and shuffles and thus only shows a small speedup. The Intel Nehalem CPU (SSE 4.2) does make use of the SSE 4.1
pmulld instruction but does not reach the full speedup of four because the instruction only has two cycles throughput and six cycles latency [8].

• The 16-bit integer subtraction on AMD performs much better with SSE. According to the benchmark results the scalar 16-bit integer addition has two cycles latency while the subtract instruction has three cycles latency. According to [13], both instructions should have a one cycle latency, though.

• The remaining integer operations show the expected speedup of close to a factor of four/eight.

5.2. Unit Tests

The unit tests have turned up a number of important differences in development environments that have to be handled to provide a portable library for vectorization:

• It turned out that GCC versions before 4.3 do not support the full SSE2 intrinsics API. Thus, Vc completely disables the SSE implementation for these older compilers, still fully supporting the Vc API via the scalar implementation.

• GCC 4.3 miscompiles some cases of explicitly allowed aliasing.† Most of them were worked around in the Vc library, except for one case, which was marked accordingly in the documentation and will report a warning when used.

• ICC 11.1 miscompiles the aliasing between vector and scalar types in even more places than GCC 4.3. The only possible workaround thus far is the use of the -no-ansi-alias compiler flag to disable aliasing optimizations. This, of course, makes optimization more complicated for the compiler and can lead to less optimized results.

5.3. Mandelbrot

As trivial as the explicit vectorization of the Mandelbrot algorithm seems, no compiler that was available for testing purposes was able to auto-vectorize the code. All recent versions of the GCC, ICC, and Open64 compilers were tested. GCC, for example, complains about “control flow in loop” and “Bad inner loop” for the second for-loop.

Initially, a speedup of more than an order of magnitude was observed between the Vc and builtin-types implementations. This was due to the use of std::complex: Its multiplication operator ensures correct handling of NaN and inf input values, leading to the extra overhead of a function call and several conditional jumps. In addition, the std::norm function has a much more complex implementation than required by the Mandelbrot algorithm‡.

After modification of the source code to not use std::complex§, the run-time of the Vc implementations did not change, while the non-Vc implementation reached the same speed as the Vc::Scalar implementation. The algorithm was benchmarked on the AMD Magny-Cours and Intel Nehalem systems. The binaries were created with GCC 4.5.2 (ICC generated considerably slower executables). As the measurements were done with the time stamp counter (compare Section 4.1) and the algorithm is not memory bound, the results of the Intel and AMD CPUs may be directly compared. Interestingly, the AMD CPU executes the code more efficiently, regardless of the implementation used (Figure 15). Note that Vc::Scalar executes at the same speed (or slightly faster) as the algorithm using builtin types (Figure 16). On the Intel CPU, the use of SSE did not yield a speedup as high as on the AMD CPU.

†http://gcc.gnu.org/bugzilla/show_bug.cgi?id=40141
‡std::norm(z) = \( s \cdot \sqrt{(\Re(z)^2 + (\Im(z))^2)^2} \) with \( s = \max(\|\Re(z)\|, |\Im(z)|) \); Mandelbrot only requires (the mathematically equivalent) \( \Re(z)^2 + \Im(z)^2 \). This was not an issue for the Vc types, because the general implementation of std::complex does not use these more complicated functions.

§You can find the exact code used for the measurements in the Vc repository.
6. DISCUSSION

6.1. Auto-Vectorization

As documented in compiler manuals and shown for a simple example in this paper, the auto-vectorization capabilities of compilers are limited. Most of these limitations are not going to change with most programming languages. Some programming languages have a larger freedom to rearrange data and annotate data dependencies, so that the compiler could, in theory, auto-vectorize more. However, vectorization of a given problem starts at the design of the algorithm. Once a developer has learned about SIMD operations, the incorporation into the design is straightforward. The compiler will never be able to take part in this phase. Therefore, it is important that developers get familiar with vectors/SIMD and do not let the compiler hide this hardware functionality.

6.2. API

It should have become obvious that the Vc API allows writing code that remains as readable and understandable as code that is written with the standard builtin types. This leads to an improved developer productivity when designing, implementing, and debugging vectorized code. Because of many readily available primitives useful for SIMD programming, the terseness of vectorized code is improved, which again leads to a better overview over the code for developers. Finally, the Vc types are compatible with generic APIs, such as the STL, which increases code reuse and thus again developer productivity.

6.3. Benchmarks

The results of the benchmarks show three important points:

1. With today’s x86 hardware, the failure to use SSE can result in up to a factor of 4 (or more in special cases) wasted performance. With the upcoming microarchitectures this factor will only increase.
2. Vc implements the supported operations optimally. The numbers show that it is possible for a C++ compiler to completely eliminate the syntactic sugar added by Vc.†
3. Portability to other vector units and other CPU architectures is kept, at no loss of performance.

The benchmarks for Vc were chosen to measure the latency of single (micro) operations of the classes. Thus, the real world relevance of these performance numbers has to be checked in a full application. Also, some of the comparisons of scalar versus vector implementation are slightly biased because in a real world application the scalar code can be optimized via branching where the vector code has to work with masks.

†It is ongoing work to identify the compilers and specific test cases where a C++ compiler fails to optimize correctly.
6.3.1. Peak FLOP Benchmark As can be seen in the benchmark results (Figure 9), a correctly optimizing compiler is able to reach the absolute maximum execution speed with both intrinsics and with the additional Vc classes wrapping the intrinsics. That ICC is unable to fully optimize the Vc classes away may be due to bad aliasing handling or incomplete dead store elimination in the compiler, which would explain the unnecessary stores to memory and copies.

Also, it can be seen that today’s microarchitectures allow a four-fold speedup with single-precision floating point calculations. Only a few years ago (such as with the K8 or Pentium 4 CPUs) this maximum speedup was only half as high and therefore often only useful for special applications.

The scalar code can be auto-vectorized, but then the instruction-level parallelism is lost which leads to the low gain compared to a non-vectorized version. Only if the data-parallelism is increased to processing four times as many values can auto-vectorization fully reach the peak performance. But then the same code should only be used for target platforms with a vector width of four as can be seen from the results when auto-vectorization is disabled in the compiler. This is different with Vc because it abstracts the vector width. Thus, the data parallelism is partitioned more naturally for the target platform.

6.3.2. Memory I/O It was shown that the use of vector loads and stores can provide a significant higher throughput between the caches and the registers. Thus, a high rate of calculations can be matched by the load and store units. For the case of large array sizes, the use of explicit prefetch instructions can hide some of the latency and thus provide the high throughput in the places where it is needed.

6.3.3. Arithmetics The data presented makes it clear that when a problem can be properly vectorized with single precision floats, a speedup of factor four is possible. With AVX this is supposed to become a speedup of factor eight. The speedup of factor eight is also possible today when arithmetics with 16-bit integers can be vectorized.

Since the instructions that are used for the SSE arithmetics have the same latency and throughput as their scalar variants, there is no decrease in possible superscalar execution and pipelining. Care must be taken that the transformation of the problem into a vectorized implementation does not remove this parallelism in the code and thus just transforms superscalar and pipelined execution into SIMD. The simple scalar product example in Figure 3 shows this problem. Often a vertical vectorization results in a reduction of pipelined execution and together with a little overhead to convert to and from vector registers can even make the vectorized implementation slower than the scalar variant. Therefore, in many cases the horizontal vectorization is to be preferred.

6.4. Mandelbrot

The Mandelbrot example shows how a simple example can be vectorized at a considerable gain in execution time, while compilers are principally unable to perform this task. In this case, the failure to auto-vectorize is connected to the innermost loop, which makes the Mandelbrot algorithm a Class 2 problem.

The example also showed a greater speedup on AMD CPUs than on Intel CPUs. This result should be further investigated on different problems to see whether this is an architectural feature of AMD CPUs or just a side effect of this specific algorithm.

Furthermore, the example code shows that code readability of explicitly vectorized code does not suffer much. The code got only slightly harder to understand. This is due to the parallel computation of several pixels in one go. Once a developer has learned the concept (and API) of masked operations, the API is not the limiting factor anymore.

ACKNOWLEDGEMENTS

This work was supported by GSI Helmholtzzentrum für Schwerionenforschung and the Hessian LOEWE initiative through the Helmholtz International Center for FAIR (HIC for FAIR). We would like to thank the GCC team for helpful responses to questions and bug reports, and our colleagues for many fruitful discussions. We thank Jan de Cuveland, Kristine Eschedor, and David Rohr for their thorough review of this paper.

References