An Evaluation of Current SIMD Programming Models for C++

Angela Pohl Biagio Cosenza Mauricio Alvarez Mesa Chi Ching Chi Ben Juurlink
TU Berlin, Berlin, Germany
{angela.pohl, cosenza, mauricio.alvarezmesa, chi.c.chi, b.juurlink}@tu-berlin.de

Abstract

SIMD extensions were added to microprocessors in the mid '90s to speed-up data-parallel code by vectorization. Unfortunately, the SIMD programming model has barely evolved and the most efficient utilization is still obtained with elaborate intrinsics coding. As a consequence, several approaches to write efficient and portable SIMD code have been proposed. In this work, we evaluate current programming models for the C++ language, which claim to simplify SIMD programming while maintaining high performance.

The proposals were assessed by implementing two kernels: one standard floating-point benchmark and one real-world integer-based application, both highly data parallel. Results show that the proposed solutions perform well for the floating point kernel, achieving close to the maximum possible speed-up. For the real-world application, the programming models exhibit significant performance gaps due to data type issues, missing template support and other problems discussed in this paper.

Keywords SIMD, vectorization, C++, parallel programming, programming model

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

1. Introduction

Single Instruction Multiple Data (SIMD) extensions have their origin in the vector supercomputers of the early '70s and were introduced to desktop microprocessors around twenty years later, when the demand for more compute power grew due to increasingly popular gaming and video applications. They exploit Data Level Parallelism (DLP) by executing the same instruction on a set of data simultaneously, instead of repeating it multiple times on a single, scalar value. An example is the brightening of a digital image, where a constant value is added to each pixel. When using SIMD, a vector of pixels is created and the constant is added to each vector element with one instruction.

The number of bits that can be processed in parallel, the vector size, has been growing with each SIMD generation. A short overview of the evolution of the most common SIMD extensions can be found in [14, 15]. Along with the vector size, the number of available SIMD instructions has been increasing as well, adding more advanced functions to the repertoire over the years.

In order to use these SIMD features, code has to be vectorized to fit the underlying hardware. There are several ways to accomplish this. The least effort for the programmer is using the auto-vectorization capabilities of modern compilers.

These compilers implement one or more automatic vectorization passes, commonly applying loop vectorization and Super-word Level Parallelism (SLP). Within such a pass, the compiler analyzes the code by looking for instructions that profit from the scalar to vector conversion, and then transforms it accordingly. For traditional automatic loop vectorization [6], this approach succeeds for well-defined induction variables and statically analyzable inter- and intra-loop predictions, but it fails to vectorize codes with complex control flows or structured data-layouts. SLP vectorizers [5] typically work on straight-line code and scan for scalars that can be grouped together into vectors; recent studies have shown, however, that the average number of vector lanes that are occupied is merely 2 [9].

Significantly more effort has to be spent when a programmer chooses to use intrinsic functions to vectorize code. Intrinsics are low level functions that implement all SIMD instructions of a processor architecture. Though they are at a higher level of abstraction than assembly programming, coding with them comes with its own challenges, i.e. effort, portability and compatibility. Nonetheless, intrinsics coding is still considered state-of-the-art for maximum performance gain, as its low-level approach results in efficient hardware utilization.

With these two options, SIMD programming presents itself as a trade-off between effort and performance, where the programmer can only chose between the two extrema. This is illustrated in Figure 1, where an HEVC (H.265) video decoder [16] was vectorized manually with intrinsics and compared to the results of the most commonly used C++ compilers' auto-vectorizers. It is apparent that high programming effort results in high performance and vice versa. To find a middle ground, researchers have worked on inventing more convenient programming models that still deliver sufficient performance. In this paper, we provide an overview of such programming models for current SIMD units and evaluate how well they do in terms of speed-ups. For this purpose, two kernels were assessed:

1. the Mandelbrot benchmark, which is highly data-parallel, straight-forward to vectorize and works with 32bit floating point numbers
2. an HEVC interpolation kernel, taken from a real-world HEVC decoder, also highly data-parallel, based on function templates and working with 8bit and 16bit integers.

This paper will discuss the challenges of intrinsics programming in more detail in Section 2, as the common goal of all proposed programming models is overcoming them while maintaining high performance. It will then present an overview of all proposed ap-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

WPMVP '16, March 13 2016, Barcelona, Spain.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4060-1/16/03…$15.00.
http://dx.doi.org/10.1145/2870650.2870653

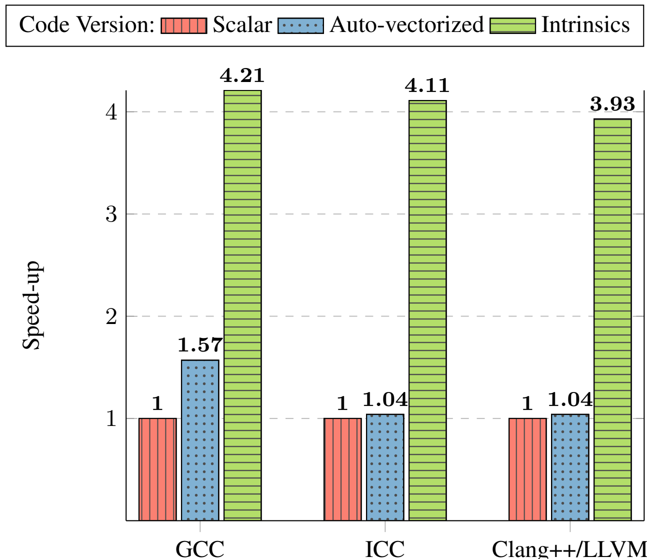

Figure 1. Speed-ups obtained with an auto-vectorized and an intrinsics-based implementation of a real-world HEVC video decoder , shown for the most popular C++ compilers *(4K resolution video decoding, 8 threads on an Intel i7-4770 core with AVX2)*

proaches in Section 3, highlighting the different paths taken by each of them. Afterwards, the kernel implementations are evaluated in Section 4. Related work is briefly presented in section 5 and all findings are summarized in Section 6.

2. Challenges of Intrinsics Programming

2.1 Effort

The biggest drawback of intrinsics coding is the significant programming effort that is required. Due to their low level of abstraction, intrinsics do not offer advanced functions or data structures. Instead, they work with elemental numeric types, such as `int` and `float`, and most of their functionality is limited to basic arithmetic, boolean and load/store operations. That is why code vectorization with intrinsics is time-consuming, because all high level language constructs, such as containers or functors, have to be mapped to this low-complexity feature set.

2.2 Portability

Besides the significant effort of programming with intrinsics, portability of intrinsics code across compute platforms is another challenge. Typically, intrinsics are tied to a specific base ISA. For example, it is not possible to run code written for Intel AVX2 on an ARM machine that supports NEON. Therefore, it is crucial to maintain several versions of a program in order to facilitate the most common processor architectures. Otherwise, only a limited set of intrinsics that is available across SIMD units can be used, ignoring each hardware's unique features and thus opportunities for performance gain.

2.3 Forward Compatibility

In addition to the portability issue, there is compatibility. Since SIMD extensions are backwards compatible in general, intrinsics code will still work when a new SIMD ISA is released. Unfortunately, it will not make use of the hardware's innovations, unless a new version using the latest intrinsic set is implemented. On the other hand, when an application provider wants to ensure maximum performance even (or especially) on older platforms, code versions have to be provided for older SIMD ISAs along with current ones, removing intrinsics added in later SIMD releases. In the end, several code versions need to exist to ensure portability and compatibility, each of them requiring time-consuming low-level coding with intrinsics.

3. Programming Models for Code Vectorization

To address these challenges of intrinsics coding, new programming models have been developed to facilitate a higher level of abstraction for code vectorization, while aiming for the same performance. This section provides an overview of such programming models, and categorizes them by their vectorization techniques. We make the distinction between implicit and explicit approaches, where implicit vectorization is performed by the compiler based on directives, and explicit vectorization requires manual code adjustments by the programmer. In addition, there are hybrid forms that combine the two concepts.

3.1 Implicit Vectorization

When using programming models based on implicit vectorization, the programmer does not have to vectorize the code manually. Instead, directives, for example in the form of pragmas, are provided to improve a compiler's auto-vectorization passes. This approach is based on Single-Program Multiple-Data (SPMD) programming models, which are typically used for multi- and many-core platforms to achieve thread-level parallelism.

When a new SIMD generation is published, this vectorization technique does not require to adjust an application source code, as parallel regions have not changed. Instead, the compiler, who interprets the directives and performs the intrinsics mapping, will have to be updated to support the new hardware features.

3.1.1 Auto-vectorization

Compiler-based automatic vectorization relies on two main techniques: loop vectorization and SLP. For this work, we assessed the vectorization capabilities of three widely used compilers.

The GCC vectorizer is automatically turned on with the `-O3` compiler flag. It includes analyses for memory access patterns and loop carried dependencies. Based on their findings, the vectorizer performs a profitability estimation to decide if a loop should be vectorized. In addition, SLP is implemented through basic block optimizations.

LLVM contains two vectorizers for loop and SLP vectorization, respectively; it is also able to detect special cases, such as unknown trip counts, runtime checks for pointers, reductions and pointer induction variables.

While ICC's vectorization approach is not published, additional documentation to help programmers with writing auto-vectorizable code is available [1]. Vectorization is typically not performed for non-contiguous memory accesses and data dependencies, though.

In addition, all three compilers offer pragma directives to provide hints for the vectorizers, for example

```
#pragma ivdep
```

in ICC. For the results in Figure 1, however, we did not apply these to the scalar code, because our goal was to test the standard automatic approach.

3.1.2 OpenMP 4.0

Open Multi-Processing [17] is an API that has been developed and enhanced by a consortium of compiler and hardware companies since the late '90s. It relies on pragmas to identify parallel regions

within the source code, and is supported by the most popular C++ compilers. While originally designed to support multi-threading, the pragma

```
#pragma omp simd
```

has been added with the latest standard version, version 4.0. It identifies functions and loops as data-parallel, and can be used to define linear memory access patterns and safe vector lengths using clauses. Unfortunately, not all C++ compilers support the latest 4.0 standard yet.

3.1.3 Cilk Plus

Cilk Plus started out as a project at MIT (named Cilk then) and is now owned and developed by Intel [18]. It is a general-purpose programming language based on C/C++, and hence integrates seamlessly into respective program codes. It targets both, thread- and data-level parallelism. Cilk Plus offers a pragma

```
#pragma simd
```

similar to OpenMP 4.0, as well as language features for code vectorization, such as array range notations. It is possible to leave original sources untouched, though, and just add pragmas that are treated as comments by compilers other than the Intel C/C++ Compiler; this is critical as ICC is the only compiler that currently supports Cilk Plus.

3.2 Explicit Vectorization

The programming models for explicit vectorization rely on the programmer to vectorize code manually, using application knowledge. To simplify this process, a set of data-types and functions to facilitate a higher level of abstraction is provided by a language library. The written code is then mapped to low-level intrinsics "under the hood", i.e. the library takes care of selecting the correct intrinsic for the target platform, while the compiler performs the registry allocation. Instead of updating the application code when a new SIMD ISA is available, only the library has to be updated once to make use of the new features. With this approach, the portability and compatibility issues are not eliminated, but reduced to the minimal effort of a single code update.

The following programming models are currently available for code explicit vectorization:

3.2.1 Intrinsics

As mentioned earlier, intrinsics programming is still the state-of-the-art programming model for best SIMD performance. Though providing a higher level of abstraction as assembly programming, generating efficient code is time-consuming and comes with its own challenges as described in Section 2. For more information, please refer to the SIMD manufacturers' manuals [25, 26].

3.2.2 C++ Data Types

Standard C++ offers two different data-types to vectorize code:

- `std::vector` and `std::array`
- `std::valarray`

`std::vector` and `std::array` are container classes that are templated to accomodate any data-type and size. They come with a set of class member functions, such as iterators, size operators and modifiers. Because this type of vector/array is not intended for mathematical operations, element-wise arithmetic functions are not provided and there is no straightforward mapping of these containers to SIMD intrinsics. As a consequence, the written code has to solely rely on the compiler for auto-vectorization and there is no performance gain when compared to strictly scalar code.

In contrast, the `valarray` class was introduced to standard C++ to provide an array notation for mathematical operations, trying to port a concept from Fortran to C++. Development stalled, though, when the Standard Template Library (STL) was added in the late 90s. Again, there is no direct connection between this class and SIMD intrinsics, thus relying on auto-vectorization as well. When using structured data, this class can actually cause a slow-down for certain accessing schemes.

3.2.3 Macro Abstraction Layer

The Macro Abstraction Layer (MAL) [21] offers the lowest level of abstraction of all the programming models evaluated in this work. It was developed as a side project when vectorizing the PARSEC benchmark suite, which was done at the Norwegian University of Science and Technology (NTNU), and offers a set of macros to use in lieu of intrinsic functions. For example, a single `ADD` macro replaces the multiple intrinsic versions for SSE, AVX and NEON platforms. MAL can be easily extended for more ISAs and intrinsic functions by adding and extending macros, as the current version only contains the functions needed for the PARSEC vectorization. Unfortunately, macros that exist in a single ISA only cannot be abstracted, when these might be critical to gain optimal performance. Moreover, the programmer needs a good understanding of intrinsics programming, ideally starting from an intrinsics implementation with the goal to generalize it for a range of ISAs.

3.2.4 Vc

The Vc library, which was developed at Goethe Universitaet Frankfurt in Germany, is based on the straight-forward approach to wrap SIMD vectors and intrinsics in high-level functions with zero overhead [22]. This approach is similar to the one taken by the Macro Abstraction Layer, but targets a higher level of abstraction and avoids dealing with macros.

Contrary to other library solutions, though, programmers do not have control over the applied vector size when using Vc. Instead, only generic vector data types, combined with a set of functions and masking operations, have to be used. For example, when using 32bit floating point numbers in the form of a `float_v` vector, the library will determine how many operations the target hardware can execute in parallel and size the vector accordingly. This has the advantage that the programmer does not need to deal with hardware specifics, but when an exact vector size is needed, like in the HEVC kernel discussed in Section 4, additional masking operations have to be applied to disable vector lanes. For example,

```
Vc::float_v my_vec(src);
my_vec &= 0xFF;
```

will ensure that only the lowest 8 entries of the vector are used, although the hardware might be able to process more.

3.2.5 Boost.SIMD

Boost.SIMD [15] was developed at the Université Paris Sud in collaboration with the Université Blaise Pascal and is now hosted by Numscale; though the name suggests otherwise, it is not part of the official Boost library yet, but is incorporated into the NT^2 project. It was designed as an *Embedded Design Specific Language*, and as such uses expression templates to capture the abstract syntax tree and perform optimizations on it during the compilation stage. Boost.SIMD is based on STL containers and the definition of functors, and sources are distributed as a header-only library.

Along with an elaborate set of mathmatical functions and C++ standard components, it provides an abstraction of SIMD registers called `pack`. For a given type T and a static integral value N ($N = 2^x$), a pack encapsulates the best type able to store a sequence of N elements of type T. Such a vector is created by

```
auto my_v = boost::simd::load<pack<T,N>>(ptr);
```

When both, T and N match the type of a SIMD register, my_v will be mapped to that type; otherwise, Boost.SIMD will fall back on using `std::array<T,N>`, whose drawbacks have been discussed earlier. With this approach, it is possible to either define vector sizes when an optimal solution is known, or to leave it up to the compiler to find a mapping.

3.2.6 Generic SIMD Library

Another approach is pursued by the Generic SIMD (gSIMD) Library, developed at the University of Illinois at Urbana-Champaign in collaboration with IBM Research [20]. Similar to the implicit programming models, it applies the SPMD concept to SIMD programming, i.e. it works with fixed-lane vectors instead of fixed-width vectors. So rather than calculating how many operations of a certain data type can be executed in parallel on a target hardware, the programmer now only defines the number of execution lanes that would best fit the kernel. The library then takes care of mapping these lanes to the underlying SIMD vector sizes and hardware.

At the current state of development, the Generic SIMD Library supports Intel's SSE4.2, as well as the VSX instructions for IBM's POWER7 processor. An implementation for AVX is not yet available, as well as support for vectors with more or less than 4 lanes.

3.2.7 Cyme

The Cyme library [19], developed at EPFL in Switzerland, approaches SIMD programming at the highest level of abstraction out of all explicit programming models presented. It focuses on an application's native data structures, instead of fitting a kernel into given vector data-types. Hence, a set of containers is offered that implement Array of Structures (AoS) and Array of Structures of Arrays (AoSoA) data layouts, where most other solutions support Structure of Arrays (SoA) layouts only. Vector data-types are also available.

With this approach, Cyme targets a class of applications that is based on a large number of small but similar kernels, as often used in high-performance computing. That is why the less common QPX SIMD extension for IBM's Blue Gene /Q is supported besides Intel's SSE and AVX. Furthermore, Cyme understands itself as complimentary to other libraries, such as Boost.SIMD and VC, and the authors see the opportunity of integrating with these, since its mapping solution could be added as an enhancement for this special class of application.

3.3 Hybrid Solutions

To get the best of both worlds, hybrid solutions use a combination of implicit and explicit parallelization.

Basically, the programmer performs an explicit vectorization first by writing functions with a specific language or language extension. Afterwards, the programming model's compiler performs an implicit vectorization during the compile stage of the program.

As these languages/language extensions are based on C/C++, the code integrates seamlessly into an existing C/C++ project.

3.3.1 ispc

ispc is a research project by Intel and also pursues the idea of porting the SPMD programming model to CPUs by utilizing their SIMD capabilities [28]. So instead of running multiple program instances on multiple processors, program instances are mapped to individual SIMD lanes. Staying consistent with the SPMD approach, this allows for different conditionals and control flows for each lane.

To use ispc, the programmer has to write the application's data parallel portions as functions in ispc's own language. The remaining project is then compiled with its standard compiler, while the ispc functions are compiled with the specific ispc compiler, based on LLVM. As ispc's syntax and semantics are derived from C/C++, the functions integrate seamlessly into a C/C++-based project during the linking stage.

```
void Mandel(float x1, float y1,
            float x2, float y2,
            int width, int height,
            int maxIters, int * image)
{
  float dx = (x2-x1)/width;
  float dy = (y2-y1)/height;

  for (int j = 0; j < height; ++j){
    for (int i = 0; i < width; ++i){

      MyComplex<float> c(x1+i*dx, y1+j*dy);
      MyComplex<float> z(0,0);
      int count = -1;

      while ((++count<maxIters) && (norm(z)<4.0))
        z = z*z+c;

      *image++ = count;
    }
  }
}
```

Figure 2. Scalar implementation of the Mandelbrot benchmark, using a slimmed-down complex numbers class for performance reasons

For this programming model, it is crucial that the parallelizable regions of code can be separated from scalar code. Otherwise, it is not possible to use the ispc compiler for the SIMD lane mapping. Also, ispc does not support all C/C++ language features, for example templated functions cannot be resolved due to the two compilers used for scalar and parallel code.

3.3.2 Sierra

Sierra, developed at Saarland University, Germany, adds the new type constructor `varying(L)` to the C++ language [27]. Acting as a type qualifier syntatically, the programmer can define vectors of different data-types and parameterizable size. A vector containing four integers, for example, is created with

```
int varying(4) = {1,2,3,4};
```

In this programming model, it is the programmer's responsbility to ensure a vector size that matches the underlying hardware, for example by examining pre-defined compiler macros.

Conveniently, Sierra provides automatic masking of vector lanes when a vector conditional is evaluated. Thus, writing code with Sierra is very close to writing scalar code, except for variable declarations.

In order to build the vectorized sources, Sierra's own compiler is needed to resolve the new data type; the compiler is based on Clang++/LLVM.

4. Performance Evaluation

4.1 Mandelbrot Set

4.1.1 Experimental Setup

The first kernel that is used to test the proposed programming models is the Mandelbrot benchmark. This standard kernel calculates the Mandelbrot set, a set of complex floating-point numbers, and is highly data-parallel. A scalar implementation is shown in Figure 2, where each set member is calculated as a pixel of a graphical representation. With each member being independent from its neighbours, the inner `for`-loop can be fully vectorized. Hence, it should be possible to achieve a compute platform's maximum SIMD speed-up, given that the iteration count of the internal `while`-loop is large enough to hide memory access times.

In this work, an Intel i7-4770 processor supporting AVX2 was used for all measurements. Thus, a speed-up of up to 8x should be feasible, since AVX2 supports vector sizes of 8 x 32bit floating-point numbers.

The Mandelbrot kernel was either provided with the programming model's sources as an example, or implemented by the authors themselves, with the exception of the intrinsics version [23]. The Macro Abstraction Layer was left out of this evaluation, as the provided functions were not sufficient to implement the kernel, and enhancing the library would have been necessary before assessing the prospective performance gain.

Speed-up was measured by comparing the number of execution cycles to the auto-vectorized baseline from Figure 2. The execution cycles were measured using the infrastructure delivered with the ispc examples [29], which is based on standard `rdtsc()` functions. All results — with the exception of Sierra — were generated using ICC, but the GNU Compiler Collection and Clang++/LLVM produced the same quality of results with similar execution times.

4.1.2 Results

Figure 4.1.2 depicts the speed-ups obtained by the different programming models.

As expected, the proposed programming models perform well and a significant speed-up is achieved by most. The only two exceptions are gSIMD, compiled for AVX2, and Cyme. For gSIMD, it has to be taken into consideration that this library only supports SSE4.2 with a maximum vector size of 128bit for floating point numbers, i.e. a 4x maximum speed-up. When benchmarking all programming models using SSE4.2, gSIMD is on the same level as other programming models, achieving a 1.9x – 2.3x speed-up.

For Cyme, the data-layout in the Mandelbrot kernel does not match the library's AoS or AoSoA containers. What happens is that the kernel has to be adapted to these data structures, and as a consequence, performance goes down due to non-native data reordering. In short, this library was not designed for this application class. As the HEVC kernel is similar to the data layout in this example, this library will not be evaluated for the next example.

All other programming models perform well, some even outperforming the AVX intrinsics implementation. For Boost.SIMD and Vc, the effort of explicitly vectorizing the code paid off, as they achieve an even higher speed-up than the implicit proposals Cilk Plus and OpenMP 4.0. It is also interesting to note that vectorization is beneficial even for a single or small numbers of loop iterations.

4.2 HEVC-Kernel

4.2.1 Experimental Setup

The second kernel that was vectorized is part of the HEVC decoder benchmarked in Figure 1. In this kernel, a filter is applied to a video frame, meaning that for each pixel, the dot-product of the pixel's neighbours with a set of coefficients is calculated. Afterwards, the result is clipped to fit into the $[0, 255]$ interval and stored in a separate array.

The kernel function is templated to work with unsigned 8bit or 16bit integer pixels, i.e. data-types `uchar` and `short` in C++, as well as different configuration options. For example, a template parameter determines the number of neighbours used for calculating the dot-product. A stride parameter for vector gathering is introduced, too, to enable vertical and horizontal filtering. No further control flow is added within the loops. The basic kernel code is shown in Figure 4.

```
template<typename Tsrc, bool shiftBack, ...>
void interpol(const Tsrc* src, short *dst, const short* coeff,
        int srcStride, int dstStride,
        int width, int height, int shift, ...)
{
        ...
        // some parameter calculation
        ...

 for (int row = 0; row < height; row++) {
   for (int col = 0; col < width; col++) {

     int sum = 0;

     for(int i = 0; i < size; i++){
       sum += src[col + srcStride*i] * coeff[i];
     }

     if (shiftBack){
       dst[col] = MyClip(0, iMaxVal, (sum + offset) >> shift);
     } else {
       short val = sum >> shift;
       ...
       // more store options
       // with templated control flow
       ...
     }
   }
   src += srcStride;
   dst += dstStride;
 }
}
```

Figure 4. Scalar source code of the HEVC interpolation kernel

We implemented the interpolation kernel with the proposed programming models, based on knowledge from published papers, examples and documentation. Results were measured with the same system used for the Mandelbrot evaluation, an Intel i7-4770 processor with AVX2. As the performance is workload dependent, i.e. dependent on the video resolution, encoding scheme etc., a set of 104 standard test videos was benchmarked.

The kernel's maximum theoretical speed-up depends on the operand sizes of the dot product. As one of the vectors is always of data type `short` and AVX2 supports integer vectors with up to 128bit, the speed-up is limited to 8x.

4.2.2 Results

Figure 4.2.2 shows the speed-ups obtained for the best and worst test-video, referring to the videos achieving highest and lowest speed-up when using the intrinsics implementation. Execution times were measured by a built-in program timer.

As can be seen in this figure, the programming models could not reach intrinsics performance by far, contrary to the results of the Mandelbrot experiment. Even worse, some implementations were slowing down the kernel.

At this point it must be mentioned, though, that the intrinsics implementation also applies saturated arithmetics for clipping and shifting operations that are performed before storing the data. That is why a speed-up of more than the theoretical maximum of 8.6x was achieved. Although programming models like ispc offer this kind of arithmetic functions, they were not not applied to the kernel implementations; the reason being that the vectorization capabilities were to be tested, and the saturated arithmetics only make up a small portion of the overall performance gain.

A shared problem among several programming models, i.e. gSIMD, Cyme and ispc, was an issue with data-type promotion. In the HEVC kernel, a multiplication of a `short` and a second operand, either `uchar` or `short`, is performed as part of the dot product. Although precautions were taken by the programmer and the result was assigned to an `int` variable, overflows occurred, whereas the scalar version yielded correct results in accordance to the language standard. This is illustrated in Figure 6. As a consequence, all vectors had to be promoted to `int`, reducing the maximum possible speed-up factor by 2.

Boost.SIMD was not able to resolve the kernel to a faster implementation than scalar code, and neither a performance loss nor gain was obtained. Apparently, the `pack` data types were resolved to `std::array` instead of SIMD vector types. A possible contribut-

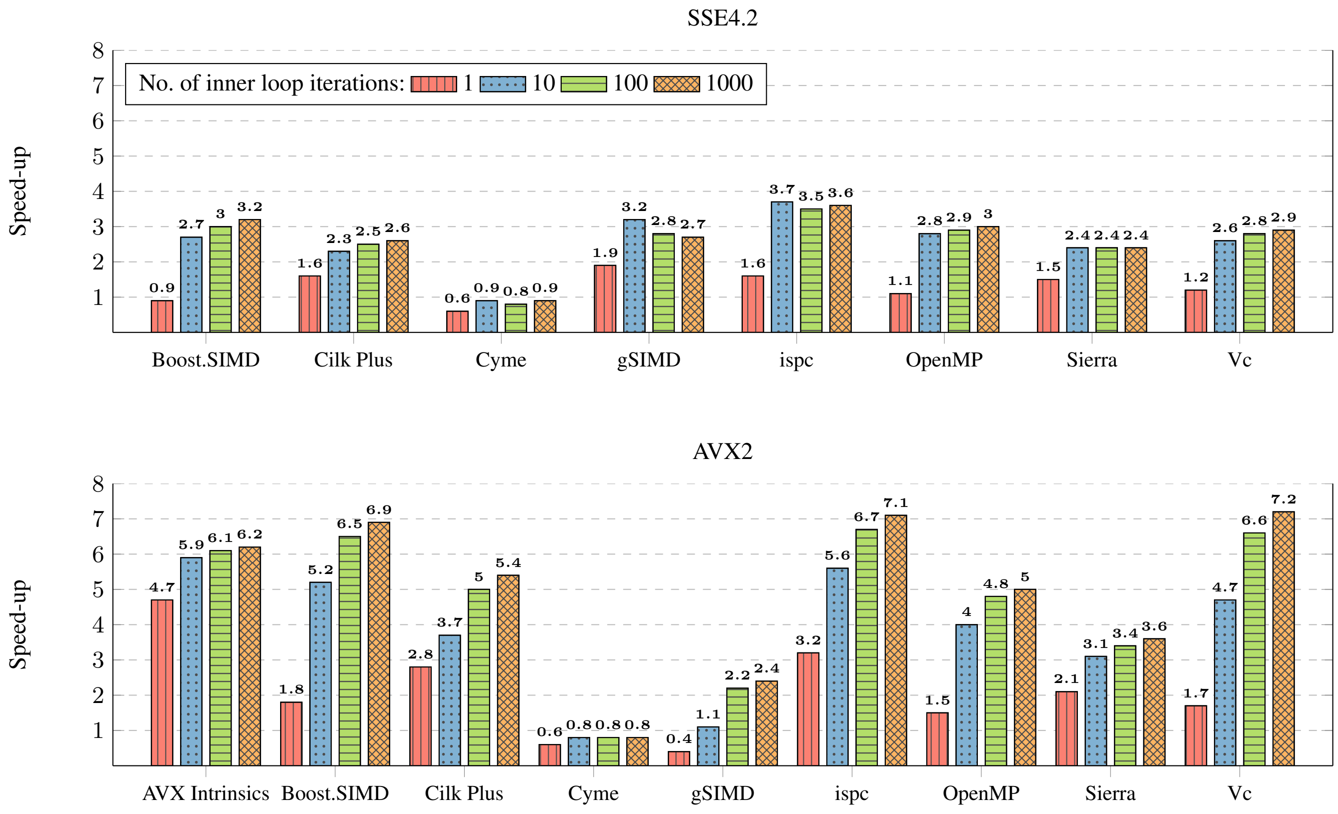

Figure 3. Acceleration of a Mandelbrot kernel with SSE4.2 and AVX2 SIMD units, shown for varying iterations of the inner `while`-loop

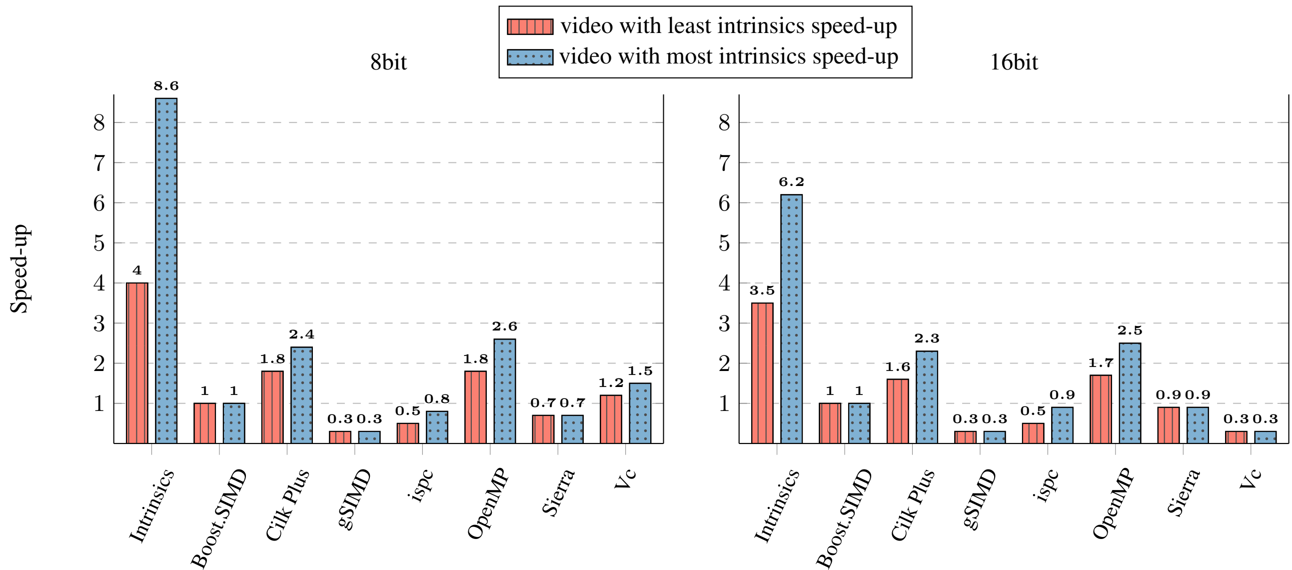

Figure 5. Speed-Ups of the HEVC interpolation kernel, showing best and worst test-video results for 8bit and 16bit pixel-depth

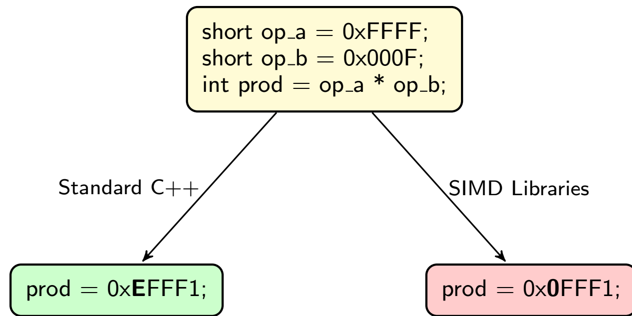

Figure 6. Problem of data-type promotion in gSIMD, Cyme, and ispc

ing factor were the manual gather implementations, as the ones provided with the library were either not implemented or documented properly.

The gSIMD implementation was affected by the library's level of maturity, because the current version only supports vectors with 4 lanes. But, in the HEVC kernel, either 4 or 8 neighbors of a pixel are accessed to calculate the dot product. When 8 neighbors are needed, the calculation had to be looped twice to produce correct results, so the inner for-loop was not eliminated.

The different number of neighbours also played a role in the Vc implementation. Since Vc does not allow the programmer to determine the vector size, masking precautions had to be put in place. Nonetheless, performance is best for 8bit pixel depth out of all explicit programming models. It drastically decreases for 16bit pixel depths, though. The big difference here is the gathering of vector data. For this kernel, two different versions had to be implemented, as Vc does not support 8bit integers. When gathering 8bit source data into vectors, a built-in gather function could be used for a stride of 2, otherwise a scalar implementation including a type cast to `short` was used as a fall-back solution. In the 16bit version, the built-in gather functions were used exclusively. Apparently, this had a significant performance impact and the kernel slows down significantly.

Besides the data type promotion issue, ispc faced a problem due to the heavy templating of the original scalar function. Since template parameters had to be handed to ispc as function arguments, they could not be resolved at compile time and a constant value could not be assumed by the compiler. For example, the shift amount for the store operation caused performance warnings during compilation, which was not an issue in the scalar function as this parameter is always a power of two. When all template parameters were treated as constants in the ispc code and the failing data type promotion was ignored, a speed-up of 50% could be achieved in a small additional experiment.

For the Sierra implementation, the biggest problem was the lack of appropriate constructors and gather operations. These had to be hand-coded, because pointer support is not yet completed in the current version and all vectors had to be created based on pointers to pixel sources. Afterwards, code resembled a scalar implementation and more overhead was added than the SIMD acceleration could make up.

5. Related Work

Automatic vectorization Loop vectorization has been improved with efficient run-time and static memory alignment [2, 3], reducing the data permutation [4] and other techniques including outer-inter loop interchange and efficient data realignment [6]; however, Maleki et al. [7] showed that state-of-the-art compilers can only vectorize a small fraction of loops in standard benchmarks like *Media Bench*.

Recently, a number of publications addressed the vectorization of straight-line code through Super-word Level Parallelism (SLP). Those include SLP exploitation in the presence of control flow [10], by using SLP tree [12] and throttled graph to stop vectorization prematurely when it is not beneficial anymore [9]; an alternative approach implements SLP with a back-end vectorizer that is closer to the code generation stage, therefore with more precise information for the profitability estimation analysis [11].

OpenCL The Open Computing Language (OpenCL) [13] provides a standard interface for parallel computing using task-based and data-based parallelism. It is an open standard maintained by the Khronos Group and compliant implementations are available from different vendors, such as Altera, AMD, Apple, ARM, Intel, NVIDIA, Xilinx and others. OpenCL implementations may implicitly exploit vectorization on devices with such capability. For example, the Intel SDK for OpenCL implements an implicit vectorization module; it maps OpenCL work-items to hardware according to SIMD elements, so that several work-items fit in one SIMD register to run simultaneously.

Intel Array Building Blocks No longer maintained, Intel's Array Building Blocks (ArBB) [31] were proposed as a counterpart to Intel's Thread Building Blocks in 2011. In 2012, though, the effort has been given up in favor of pursuing Cilk Plus, and sources are no longer available [30].

The concept is based on providing an embedded, compiled language whose syntax is implemented as an API. To the programmer, it presents itself as a language library, altough the API is layered on top of a virtual machine, which implements the compilation and parallelization. Performance numbers published for Mandelbrot speed-up were between 3x – 4x, thus compliant with the numbers we show for SSE4.2 acceleration.

6. Conclusion

SIMD extensions have been introduced to microprocessors in the late 1990s, with regular releases of new ISA extensions up to this day. Nonetheless, using them efficiently still remains a problem due to the lack of a sufficient programming model. The state-of-the-art in terms of performance, i.e. intrinsics programming, comes with its challenges of effort, portability and compatibility.

Consequently, new solutions to improve productivity and hardware utilization have been proposed, focusing on abstracting code vectorization while maintaining performance. In this work, we categorized these approaches into implicit and explicit vectorization techniques; for implicit approaches, vectorization is taken care of by the compiler, while for explicit approaches, vectorization is typically done by the programmer on a higher level of abstraction than intrinsics coding.

We assessed a set of programming models by implementing two different kernels and comparing the achieved speed-ups. The first kernel calculates a graphic representation of the Mandelbrot set, and is commonly used to showcase the power of SIMD extensions. The kernel works with 32bit floating point numbers and vectorization is straight-forward. Results show that most of the proposed programming models achieve a speed-up close to the theoretical maximum. In our measurement, the explicit programming models fared slightly better than the implicit programming models.

As a second benchmark, we implemented an interpolation kernel from a real-world HEVC decoder. It calculates the dot product of each pixel's neighbors with a set of coefficients, and works on small integer data types, such as `char` and `short`. Results showed that the intrinsics implementation is at least 2x-8x faster than all

other programming models, not taking into consideration those approaches that slowed down the kernel. For this benchmark, the implicit approaches beat the explicit approaches.

What can be seen from these results is that a single proposal that improves performance, reduces implementation effort and fits all types of applications does not yet exist. Each approach has its unique advantages, and a programming model needs to be chosen carefully, knowing the application's structure and demands.

In this work, we focused on assessing highly data parallel kernels. Nonetheless, a major obstacle for failing auto-vectorizers are complex control flows and data structures within a kernel. Hence the programming models' capabilities to deal with such applications need to be investigated next in order to give a more thorough analysis of the currently available programming models.

References

[1] Intel. *A Guide to Auto-vectorization with Intel C++ Compilers*. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`, 2012

[2] A. E. Eichenberger, P. Wu, and K. O'Brien. *Vectorization for SIMD architectures with alignment constraints*. Proceedings of the Conference on Programming Language Design and Implementation (PLDI), 2004

[3] P. Wu, A. Eichenberger, and A. Wang. *Efficient SIMD code generation for runtime alignment and length conversion*. Proceedings of the International Symposium on Code Generation and Optimization (CGO), 2005

[4] G. Ren, P. Wu, and D. Padua. *Optimizing data permutations for SIMD devices*. Proceedings of the Conference on Programming Language Design and Implementation (PLDI), 2006.

[5] Samuel Larsen and Saman Amarasinghe. *Exploiting Superword Level Parallelism with Multimedia Instruction Sets*. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2000

[6] Dorit Nuzman and Ayal Zaks. *Outer-loop Vectorization: Revisited for Short SIMD Architectures*. Proceedings of the 17th International Conference on Parallel Architectures and Compilation Technique (PACT), 2008

[7] S. Maleki, Y. Gao, M. J. Garzaran, T. Wong, and D. A. Padua, *An evaluation of vectorizing compilers*. Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), 2011.

[8] Christoforos E. Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanovic, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Randi Thomas, Noah Noah and Katherine Yelick. *Scalable Processors in the Billion-Transistor Era: IRAM*. IEEE Computer, 1997

[9] Vasileios Porpodas and Timothy M. Jones. *Throttling Automatic Vectorization: When Less Is More* . 24th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2015

[10] J. Shin, M. Hall, and J. Chame, *Superword-level parallelism in the presence of control flow*, Proceedings of the International Symposium on Code Generation and Optimization (CGO), 2005.

[11] R. Barik, J. Zhao, and V. Sarkar. *Efficient selection of vector instructions using dynamic programming*. Proceedings of the International Symposium on Microarchitecture (MICRO), 2010

[12] J. Liu, Y. Zhang, O. Jang, W. Ding, and M. Kandemir. *A compiler framework for extracting superword level parallelism*. Proceedings of the Conference on Programming Language Design and Implementation (PLDI), 2012

[13] Lee Howes. *The OpenCL Specification, Version 2.1, Revision 23*. Khronos Group, 2015

[14] N. Slingerland and A. Smith. *Multimedia Instruction Sets for General Purpose Microprocessors: a Survey*. University of California, 2000

[15] P. Estérie, J. Falcou, M. Gaunard and J. Lapresté. *Boost.SIMD: Generic Programming for Portable SIMDization*. In WPMVP 2014

[16] C. Chi, M. Alvarez-Mesa, B. Bross, B. Juurlink and T. Schierl. *SIMD Acceleration for HEVC Decoding*. In IEEE Transactions on Circuits and Systems for Video Technology, 2014

[17] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, 2013

[18] A. Robison. *Composable Parrallel Patterns with Intel Cilk Plus*. In Computing in Science and Engineering 15.2, 2013

[19] T. Ewart, F. Delalondre and F. Schuermann. *Cyme: A Library Maximizing SIMD Computation on User-Defined Containers*. In Supercomputing, 2014

[20] H. Wang, P. Wu, I. Tanase, M. Serrano and J. Moreira.*Simple, Portable and Fast SIMD Intrinsic Programming: Generic SIMD Library*. In WPMVP, 2014

[21] J. Cebrian, M. Jahre and L. Natvig. *Opimized Hardware for Suboptimal Software: The Case for SIMD-aware Benchmarks*. In ISPASS, 2014

[22] M. Kretz and V. Lindenstruth. *Vc: A C++ library for explicit vectorization*. In Sofware – Practice and Experience, 2012

[23] Intel Corp. *Introduction to Intel Advanced Vector Extensions*. `https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions`, 2015

[24] Freescale Seminconductor. *AltiVec Technology Programming Interface Manual*. `http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf`, 1999

[25] Intel Corp. *Intel Intrinsics Guide*. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`, 2015

[26] ARM Ltd. *Neon Intrinsics Guide*. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0491h/CIHJBEFE.html`, 2015

[27] R. Leißa, S. Hack and I.Wald. *Extending a C-Like Language for Portable SIMD Programming*. In PPoPP, 2012

[28] M. Pharr and W. R. Mark. *ispc: A SPMD Compiler for High-Performance CPU Programming*. In InPar, 2012

[29] Intel Corp.*Intel SPMD Program Compiler - A Simple ispc Example*.`https://ispc.github.io/example.html`, 2015

[30] Intel Corp. *Intel Array Building Blocks*.`https://software.intel.com/en-us/articles/intel-array-building-blocks`, 2015

[31] C. Newburn et. al. *Intel's Array Building Blocks: A Retargetable, Dynamic Compiler and Embedded Language*. In CGO, 2011