Extending OpenMP SIMD Support for Target Specific Code and Application to ARM SVE

Jinpil Lee[1(✉)], Francesco Petrogalli[2], Graham Hunter[2], and Mitsuhisa Sato[1]

[1] RIKEN Advanced Institute for Computational Science, Kobe, Japan
{jinpil.lee,msato}@riken.jp
[2] ARM Ltd., Cambridge, UK
{Francesco.Petrogalli,Graham.Hunter}@arm.com

Abstract. Recent trends in processor design accommodate wide vector extensions. SIMD vectorization is more important than before to exploit the potential performance of the target architecture. The latest OpenMP specification provides new directives which help compilers produce better code for SIMD auto-vectorization. However, it is hard to optimize the SIMD code performance in OpenMP since the target SIMD code generation mostly relies on the compiler implementation. In this paper, we propose a new directive that specifies user-defined SIMD variants of functions used in SIMD loops. The compiler can then use the user-defined SIMD variants when it encounters OpenMP loops instead of auto-vectorized SIMD variants. The user can optimize the SIMD performance by implementing highly-optimized SIMD code with intrinsic functions. The performance evaluation using a image composition kernel shows that the user can optimize SIMD code generation in an explicit way by using our approach. The user-defined function reduces the number of instructions by 70% compared with the auto-vectorized code generated from the serial code.

Keywords: OpenMP · SIMD vectorization · VLA programming - Vector Length Agnostic programming

1 Introduction

Recent trends in processor design accommodate wide vector extensions and many-core architectures. We expect that these trends will continue to improve the flops per watt ratio. Current Intel Xeon Phi processors have the 512-bit vector instruction set, Advanced Vector eXtensions (AVX-512), and more than 60 cores. ARM released a new vector instruction set for high performance computing, named Scalable Vector Extension (SVE) [2], which allows up to 2048-bit wide vector registers. Parallel programming is getting more important when using these architectures to exploit the potential performance. OpenMP (OMP) is widely used to describe node-level parallelism on shared-memory architectures. The directives such as *parallel* and *for* can describe thread-level parallelism on many-core architectures.

B.R. de Supinski et al. (Eds.): IWOMP 2017, LNCS 10468, pp. 62–74, 2017.
DOI: 10.1007/978-3-319-65578-9_5

On the other hand, SIMD vectorization has been done automatically by compilers. Compilers analyze code structures such as loop statements and find parallelism. When the target structures are safe to be vectorized, the compiler generates SIMD instructions. The latest OMP specifications provide new directives which help this auto-vectorization process. The *simd* directive specifies vectorizable loops (SIMD loops) in the serial code. The *declare simd* directive can be given to function definitions in the serial code to specify that the target functions are vectorizable in the SIMD loop. These directives ensure that target constructs are safe to be vectorized so that compilers can skip some hard analysis such as pointer alias analysis and avoid generating runtime checks to prevent aliasing.

The OMP directives reduce the burden of compiler analysis for SIMD vectorization. However, it is hard to optimize the SIMD code performance in OMP since the target SIMD code generation mostly relies on the compiler implementation. In this paper, we propose a programming interface connecting user-defined SIMD functions and SIMD loops. To this end, we introduce the *alias simd* directive which specifies the user-defined SIMD variant of the target function. The compiler uses the specified function in the SIMD loop instead of vectorizing the target function. By using this interface, we can split the loop iteration translation and the SIMD code generation for the loop body. Code translation for controlling loop iterations remains architecture-independent by using the OMP *simd* directive. The user can write highly optimized SIMD code with architecture-dependent programming methods such as intrinsic functions.

The main target architecture of our proposal is ARM SVE. SVE is a vector length agnostic instruction set. Most instructions use a predicate mask. Our proposal includes a way of handling predicate masks and optimization in the SVE instruction set. We also consider fixed-length vector instruction sets such as Intel AVX to make the proposal to cover traditional SIMD extensions. The user-defined SIMD function can be implemented in the various ways since our proposal only relies on the function declaration and the vector ABI. In this paper, we use intrinsic functions provided by processor vendors to implement SIMD variant functions.

The rest of the paper is organized as follows: Sect. 2 shows related works proposing explicit SIMD programming models. In Sect. 3, we briefly introduce the new ARM vector instruction set, SVE, and its intrinsic functions as preliminary knowledge. In Sect. 4, we introduce the *alias simd* directive in OpenMP, which allows explicit SIMD implementations in OMP SIMD loops. In Sect. 5, some sample code and preliminary results of the performance evaluation are given to show the effectiveness of our proposal. Finally, we discuss the future work and conclude the paper in Sect. 6.

2 Related Work

There have been many attempts to establish an explicit SIMD programming model [11]. ARM C Language Extensions (ACLE) [1], which is only available

on ARM architecture, provides a type-generic interface to program ARM SVE instructions. Thanks to its vector-length agnostic design, the iteration of the SIMD loop can be controlled without considering target architecture's vector length. ispc (Intel SPMD Program Compiler) [10] defines new programming language to describe SIMD-level parallelism. It covers various Intel SIMD instruction sets such as SSE, AVX, and AVX-512 in the Xeon Phi architecture. The language-based approach such as ispc, Intel array notation [7], Sierra [8], and Terra [3] require a dedicated implementation in compilers. [4] takes compiler-independent approach using C++ template. Cyme [5] and Vc [6] are implemented as a library. While these models provide high-level interface for SIMD programming, they assume a fixed vector length so that the SIMD loop iteration step should be modified manually when targeting an extension with a different vector register size.

3 Overview of ARM Scalable Vector Extension

SVE is a new vector extension to the A64 instruction set of the ARMv8-A architecture designed to exploit increases in hardware capability without requiring software recompilation.

The vector length in SVE can be configured dynamically in the range from 128 to 2048 bits, in multiples of 128. Although the value can be obtained through system registers, the SIMD instruction set is designed to be *Vector Length Agnostic (VLA)*.

Most instructions take a predicate register to mask available elements in the operand vector registers.

The following are some of the key features of SVE.

- 32 vector registers (Z0-Z31).
- 16 predicate registers (P0-P15).
- Configurable vector length: 128 to 2048-bit (maximum is processor-dependent).
- Enables the VLA programming model – the same program can run on machines with different vector register length, without requiring recompilation.

3.1 The Vector Length Agnostic Programming Model

Listing 1 shows an example of a vector addition in C and its equivalent SVE assembly code. The operand `p0` is a predicate register which is used to mask active and inactive lanes of the vector registers `z0` and `z1`.

```
1  ; for (i = 0; i < N; i++) { C[i] = A[i] + B[i]; }
2  ; x9, x10, x11 and x12 hold N, A, B, and C, respectively
3      mov      x8, xzr
4      b        .Lcond
5  .loop:
6      ld1d     z0.d, p0/z, [x10, x8, lsl #3]
7      ld1d     z1.d, p0/z, [x11, x8, lsl #3]
8      fadd     z0.d, z0.d, z1.d
9      st1d     z0.d, p0, [x12, x8, lsl #3]
10     incd     x8                  ; increase i
11 .Lcond:
12     whilelo  p0.d, x8, x9        ; set p0.d[i] = (i < N)
13     b.first  .loop               ; execute the loop iteration
14                                  ; if the first lane is active
```

Listing 1: Vector Add Example in SVE

Figure 1 shows how SVE instructions modify register values in Listing 1. Here, we assume that the data type of `A`, `B`, and `C` is `double *`, and the data type of `i` and `N` is `unsigned long int`.

After setting the loop induction variable (`i`, carried by `x8`) to zero, the code branches directly to the instruction `whilelo`, which compares the current iteration value `i` and the last iteration value (`N` in this case, carried by `x9`). The instruction sets the loop predicate register, `p0`, as `p0.d[i] = (i < N) ? 1 : 0`, for each one of the logical lanes implied by a SIMD loop iteration.

If at least the first logical lane of the predicate vector is active (`b.first`), the branch is taken back to the start of the loop.

The predicate register is then used in the loop body to mask out the inactive lanes. In Listing 1, the loads (`ld1d`), and the store (`st1d`) instruction use the predicate register to process only the active lanes, effectively removing the need

256-bit SVE

Iter	x8 (i)	whilelo p0.d, x8, x9 ($i < N$)
0	0	1|1|1|1
1	4	1|1|1|1
2	8	1|1|1|1

384-bit SVE

Iter	x8 (i)	whilelo p0.d, x8, x9 ($i < N$)
0	0	1|1|1|1|1|1
1	6	1|1|1|1|1|1

512-bit SVE

Iter	x8 (i)	whilelo p0.d, x8, x9 ($i < N$)
0	0	1|1|1|1|1|1|1|1
1	8	1|1|1|1|0|0|0|0

Fig. 1. Vector loop control using the predicate generated by the `whilelo` instruction, for different SVE implementations. `N` is 12. Notice that the same code in Listing 1 works independently on the vector size thanks to the `incd x8` instruction. In the predicate representation, logical lane numbering is intended left-to-right.

of introducing a scalar loop tail to fix up the last elements of the computation that do no fill a full vector register length.

The logical iteration of the loop is then advanced using the `incd` instruction, which is used to increase the iteration variable `i` by the number of `double` elements a scalable vector register can hold.

The, another `whilelo` instruction is issued and the branch condition in `.Lcond` is checked again.

For the interested reader, other examples showing how to use SVE for VLA programming are available in the white paper [9].

3.2 Intrinsic Programming Interface

Like most SIMD instruction sets, SVE has an intrinsic programming interface which can be used in high-level programming languages such as C and C++. ARM C Language Extensions (ACLE) has been extended to support SVE. Listing 2 shows the ACLE version of the vector addition given in Listing 1. Because of its VLA approach, the loop is written using the *while* construct. *svbool_t* is the data type for predicate registers. *svfloat64_t* is the data type for double precision FP registers.

```
1  unsigned long i = 0;
2  svbool_t p = svwhilelt_b64_s64(i, N);
3  svbool_t tp = svptrue_b64();
4  while (svptest_first(tp, p)) {
5    svfloat64_t vec_a = svld1(p, &(A[i]));
6    svfloat64_t vec_b = svld1(p, &(B[i]));
7    svfloat64_t vec_c = svadd(p, vec_a, vec_b);
8    svst1(p, &(C[i]), vec_c);
9    i += svcntd();
10   p = svwhilelt_b64_s64(i, N);
11 }
```

Listing 2: Vector Add Example in ACLE

The predicate variable is created by *svwhilelt_b64_s64()* which do the same process in Listing 1. *svptrue_b64()* generates a predicate in which all elements are active. At the beginning of every iteration, *svptest_first()* checks the head of the predicate register to see if the next iteration has an active predicate element to process. The load and store instructions in Listing 1 are equivalent to *svld1()* and *svst1()*. *svadd()* calculates SIMD addition of the double data type. *svcntd()* returns the number of 64-bit elements in a vector register. It is then used to increase the loop iteration variable for the next SIMD execution.

Note that many routines in Listing 2 are given without specifying the element data type. It is because ACLE provides a type generic programming interface implemented using templates in C++, or *_Generic* in C11.

4 Explicit Programming Interface for Vectorizing Functions

As discussed in Sect. 1, the current OMP specification cannot specify the SIMD implementation of functions used in SIMD loops. Although the directives can help the compiler check that the target code can be vectorized, the SIMD code generation is a transparent part to the user. In this section, we propose an explicit programming interface to expose user-defined SIMD functions available in OMP SIMD loops.

4.1 Overview of the Proposed Programming Model

The basic concept of our proposal is that we provide SIMD variants of existing functions instead of using auto-generated SIMD functions. To this end, we add a new directive, named *alias simd*, in the OpenMP specification. Listing 3 shows an example code of the *alias simd* directive. Vector data types (*int4_t*, *int8_t*) and intrinsic functions (e.g. *intrinsic_add4*) in the listing are pseudo code.

```
1  #pragma omp declare simd notinbranch          // A
2  int add(int a, int b) {
3    return a + b;
4  }
5  #pragma omp alias simd to(add) simdlen(4)     // B
6  int4_t add_vec4(int4_t a, int4_t b) {
7    return intrinsic_add4(a, b);
8  }
9  #pragma omp alias simd to(add) simdlen(8)     // C
10 int8_t add_vec8(int8_t a, int8_t b) {
11   return intrinsic_add8(a, b);
12 }
13
14 #pragma omp simd simdlen(VL)  // VL is 4 or 8
15 for (i = 0; i < n; i++) {
16   z[i] = add(x[i], y[i]);
17 }
```

Listing 3: Explicit SIMD Variant in OMP SIMD Loop

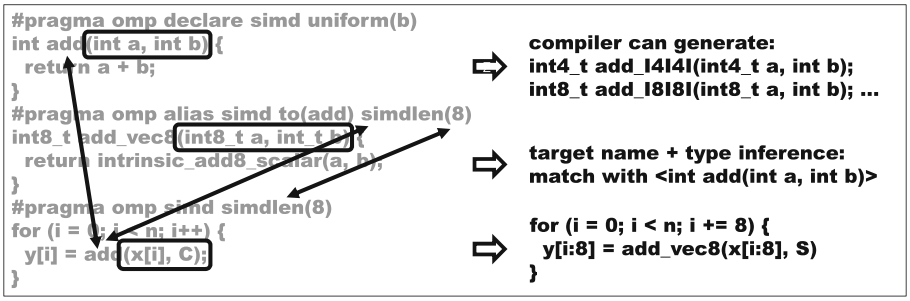

Fig. 2. Code Translation OverviewCode Translation Overview

The purpose of the *alias simd* directive is mapping SIMD variant functions defined by the user to the original functions in the serial code. In B and C in Listing 3, the function name or declaration is given by the *to* clause for the mapping process. We need the full declaration when functions with different argument types have the same name by using template (in C++) or *_Generic* (in C11). *alias simd* has the *simdlen* clause to distinguish SIMD variants by the vector length. These variants can be defined at the same time. We do not expect that Intel AVX and ARM SVE SIMD variants are available at the same time since we assume that portability among vendors is maintained by using some guard macros (e.g. *__AVX__*, *__ARM_NEON__*).

The mapping process is independent from *declare simd* so that the SIMD function is generated from the scalar function even if SIMD variants are given. SIMD variants have priority over the compiler-generated SIMD function when translating the OMP *simd* directive. The compiler will select a SIMD variant by the proper vector length (or the vector length can be given explicitly by the *simdlen* clause). We may need the scalar function definition and the *declare simd* directive in case that the loop is not vectorizable, or no proper SIMD variant is given.

Figure 2 shows how the compiler translates a OMP loop and replace functions with the SIMD variant by the *alias simd* directive. The *simd* directive specifies that the following loop should be vectorized and the vector length is 8. Function *add()* is used in the loop body. To vectorize the loop, the compiler needs to generates the SIMD code of *add()*.

The *declare simd* directive is given with the scalar code of *add()*. Since *simdlen* is not given, SIMD functions with any vector length can be generated from the compiler. In this example, the SIMD function with vector length 8 (*add_I8I8I()*) can be generated for the loop body.

On the other hand, a user-defined SIMD function is given with the *alias simd* directive. *add_vec8()* is implemented by using (pseudo) SIMD intrinsic functions. The function adds a scalar value to a vector register. On most architectures, each SIMD intrinsic function is specific to a vector length. The *simdlen* clause is given in the *alias simd* to tell the compiler that the following function can be used

to execute 8 iterations of int operations in parallel. The *to* clause specifies the original scalar function. It tells the compiler that *add_vec8()* is a SIMD variant of *add()*. The compiler infers data types of arguments in (add()) to complete the function declaration. The process follows the architecture's vector ABI. In this example, the compiler infer *int* from *int8_t*.

When the compiler translates the OMP loop in Fig. 2, two SIMD functions are available, compiler-generated *add_I8I8I()* and user-defined *add_vec8()*. In our proposal, user-defined functions have higher priority to allow the user to optimize the SIMD performance by implementing fast SIMD algorithms.

4.2 Syntax of the Alias Simd Directive

Figure 3 shows the syntax of the *alias simd* directive. The directive is given along with the complete definition of a SIMD variant. We do not assume any specific programming model for the implementation. Therefore, any kind of programming model can be used to implement SIMD variants if they adhere to the proper argument types and vector length. At first, we explain the syntax for the fixed-length SIMD architecture, and then extend it for SVE's VLA approach.

```
#pragma omp alias simd to(name_or_decl ) [clause_list ]
function_definition

name_or_decl := function_name
              | function_declaration
clause := simdlen(integer_expr )
        | inbranch
        | notinbranch
        | linear(linear_list  [: linear_step ])
```

Fig. 3. Definition of Alias Simd Directive

The *to* clause comes with either the name or declaration of the target function. When a function name is given, the compiler would infer the scalar type of each vector/scalar argument in the SIMD variant. The type reference follows target architectures's vector ABI. Multiple types can be mapped to the same vector type (e.g. generic programming model). In that case, the complete declaration should be given to choose the correct target function. The *to* clause cannot be omitted.

The *simdlen* clause specifies the SIMD length used in the SIMD variant. By the *simdlen* clause given in the *simd* directive, the SIMD loop may require several SIMD variants for the same function. The *simdlen* clause in *alias simd* is used to link the correct SIMD variant to a function call in the SIMD loop. The compiler registers the SIMD variant as the default SIMD implementation for the architecture when *simdlen* is omitted. When the target instruction set is SVE, *simdlen* is omitted by default. However, we can still use *simdlen* for SVE. This can be useful when there are highly-optimized SVE SIMD functions for a specific vector length.

The *inbranch*/*notinbranch* clause specifies whether the target function is called in a conditional statement or not. For example, the SIMD variant have additional arguments when *inbranch* is given. This clause is used to choose the correct SIMD variant, and infer the scalar types of the target argument (with *inbranch*, mask/predicate argument will be excluded in type inference).

The *linear* clause specifies the linear step of the target scalar variable increased in SIMD lanes. Regardless of the step value, the corresponding argument would have the original (scalar) data type. Since the privatization and linear increment should be implemented inside the SIMD function, multiple variants with different steps look the same from the compiler. The *linear* clause in *alias simd* should be given to distinguish the multiple SIMD intrinsic variants in the source code so that the compiler can choose the correct one. The syntax of *linear_list* and *linear_step* is the same as for the already existing OpenMP constructs.

5 Preliminary Evaluation

In this section, we introduce a use case of our proposal and perform a preliminary evaluation. We use the *alias simd* directive to optimize a simple image composition code. ALCE intrinsic functions are used to implement a SIMD function in SVE. Since the proposal has not been implemented yet, we compare the auto-vectorized code, which is equivalent to OMP SIMD vectorization in the current LLVM implementation, and the hand-written SIMD code simulating the behaviour of the *alias simd* directive. Both the serial and ACLE code are compiled by the SVE LLVM compiler and the binaries are executed on the instruction simulator, which has been provided by ARM.

5.1 Vectorization of Image Composition Kernel

Listing 4 shows the serial implementation of the composition code and the main loop. All color values are stores in the *unsigned char* type which has a range from 0 to 255. Each image has four channels, *red*, *green*, *blue*, and *alpha*. The image composition is done by a loop statement. In each iteration, function *add_filter()* is called for the *red*, *green*, and *blue* channel.

```
typedef unsigned char uchar;
typedef unsigned short ushort;

#pragma omp declare simd
uchar add_filter(uchar a2, uchar in1, uchar in2) {
  if (a2 > 0) {
    ushort temp = (ushort)in1 + (ushort)in2;
    if (temp > 255) return 255;
    else return (uchar)temp;
  }
  else return in1;
}

uchar out_r[N]; uchar out_g[N]; uchar out_b[N];
uchar in1_a[N]; uchar in1_r[N]; uchar in1_g[N]; uchar in1_b[N];
uchar in2_a[N]; uchar in2_r[N]; uchar in2_g[N]; uchar in2_b[N];

void loop() {
#pragma omp simd
  for (int i = 0; i < N; i++) {
    out_r[i] = add_filter(in2_a[i], in1_r[i], in2_r[i]);
    out_g[i] = add_filter(in2_a[i], in1_g[i], in2_g[i]);
    out_b[i] = add_filter(in2_a[i], in1_b[i], in2_b[i]); }}
```

Listing 4: Image Composition Code

add_filter() returns the sum of the two input images when the alpha mask value of the second image is not 0. When the value is 0, it returns the color value of the first image. Since the summation may overflow the maximum value (255), the code stores the temporary data in the *unsigned short* type, and checks the value range. Therefore, the serial code contains type conversion and branching.

Listing 5 shows the ACLE implementation of the composition code. Note that the function has additional argument *p*. It is because the compiler generates a predicate value to process the remainder loop as shown in Sect. 3. We assume that SVE vector functions always have a predicate variable as a first argument. It is not used in type inference shown in Sect. 4.

```
#pragma omp alias simd to(add_filter)
svuint8_t add_filter_acle(svbool_t p, svuint8_t a2,
  svuint8_t in1, svuint8_t in2) {
  svuint8_t zero = svdup_n_u8_x(p, 0);
  svbool_t alpha_mask = svcmpgt_u8(p, a2, zero); // a2 > zero
  svuint8_t temp = svand_u8_z(alpha_mask, in2, in2);
  return svqadd_u8(in1, temp);
}
```

Listing 5: Vectorization of Image Composition Code using Alias Simd

The ACLE version uses *svqadd_u8()*, saturating integer addition, to calculate the summation. When the summation is outside the range, *svqadd_u8()* ensures

that the value will be the maximum (255). This can avoid type conversion shown in the serial version and therefore increase performance. In SVE, branches can be replaced by SIMD instructions with predicate registers. The alpha mask values are checked in parallel by *svcmpgt_u8()*, which generates a predicate value. It can be used to generate the second operand of the summation to avoid the branch. The values are set to zero when the corresponding predicate value is inactive. As a result, the ACLE implementation can exploit the SIMD-level parallelism for *unsigned char* (8-bits) on the target hardware.

Listing 6 shows the main loop code written in ACLE. Even though ACLE provides a generic programming interface, porting to ACLE requires manual transformation including rewriting loops, generating predicates, and adding vector load/store instructions. As shown in Listing 4, this transformation is transparent and portable in our approach since it is programmed by the OMP *simd* directive.

```
1  void loop() {
2    int i = 0;
3    svbool_t p = svwhilelt_b8_s32(i, N);
4    svbool_t tp = svptrue_b32();
5    while (svptest_first(tp, p)) {
6      svuint8_t vin1_r = svld1_u8(p, in1_r+i);
7      // loads for vin1_{g, b}, vin2_{a, r, g, b}
8      svuint8_t vout_r = add_filter_acle(p,vin2_a,vin1_r,vin2_r);
9      svuint8_t vout_g = add_filter_acle(p,vin2_a,vin1_g,vin2_g);
10     svuint8_t vout_b = add_filter_acle(p,vin2_a,vin1_b,vin2_b);
11     svst1_u8(p, out_r+i, vout_r);
12     // stores for vout_{g, b}
13
14     i += svcntb();
15     p = svwhilelt_b8_s32(i, N); }}
```

Listing 6: Main Loop in ACLE

5.2 Evaluation Results

Figure 4 shows the performance of the auto-vectorized serial code and the hand-written ACLE code. Since we do not assume any specific hardware implementation, the performance is measured by counting the number of instructions issued during the execution of the loop. We have evaluated the performance with two dataset sizes, 32×32 and 320×320 pixels, using two simulated hardware implementations, with 256-bit and 1024-bit wide vectors.

The results show that the hand-written code simulating *alias simd* executes less instructions compared with the auto-vectorized code. In most cases, the auto-vectorized code executes $3.7 \sim 3.8$ times more instructions than the hand-written code. When increasing the vector length in the small data set (32×32 with 1024-bit SIMD), the ratio is decreased to 3.4 because the total instruction number is small and instructions for loop control becomes significant.

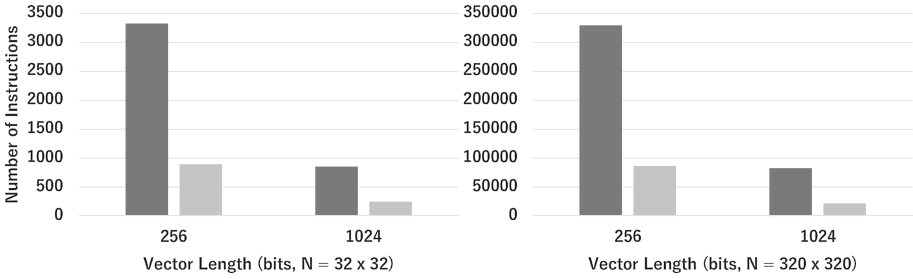

Fig. 4. Performance of Image Composition Kernel, as number of total instructions issued when executing the loop

The serial version includes the type conversion from *unsigned char* to *unsigned short* to calculate summation of two pixels. In auto-vectorization, the compiler generates SIMD addition instructions for *unsigned short*, which doubles the number of SIMD add instructions per iteration compared to the intrinsic code. Before the calculation, the compiler generates type conversion instructions. It also adds extra calculations which do not exist in the intrinsic code.

The branch used for color clamping to the maximum value is translated to SIMD compare and selection instructions in auto-vectorization. There are type conversion to *unsigned short* since the calculated values are stored in the *unsigned short* type. After the calculation, the data type is converted into *unsigned char*. The check for alpha mask is translated in the same way.

On the other hand, the intrinsic version calculates the summation using *unsigned char* type instructions. Since it uses the saturating addition instruction, *svqadd_u8()*, the range check and clamping is unnecessary. The optimization is intended to avoid the unnecessary type conversion to reduce the number of instruction executed, and improve the instruction throughput by using *unsigned char* type SIMD instructions.

It should be emphasized that our approach provides an explicit way of programming SIMD instructions. The performance result shows that our approach can successfully change the way how the code uses the SIMD instructions, which cannot be done with existing OMP SIMD directives. This is important even for a product-level compiler since it cannot always generate the optimal SIMD code. And there may be a gap between high-level languages and hardware instruction sets which make it difficult to describe the optimal SIMD code. We have used the saturating addition instruction in SVE, which cannot be described directly in the C language without using a wider type.

Our proposal is designed to be independent from instruction sets. If we implement the code transformation of *alias simd* for a specific SIMD instruction set and the vector ABI, we can describe user-defined SIMD functions to optimize the SIMD performance on the target architecture. For example, we can optimize the SIMD performance using Intel AVX intrinsic functions on Intel processors (e.g. generating a histogram using Intel AVX512-CD).

6 Conclusion

In this paper, we proposed a new OMP directive, *alias simd*. It specifies user-defined SIMD variants of functions called in SIMD loops. The compiler uses the SIMD variant when translating OMP loops instead of auto-vectorized SIMD variants. The user can optimize the SIMD performance by implementing highly-optimized SIMD variants with intrinsic functions. Even for a product-level compiler, it is difficult to generate optimal SIMD code for every case. Our proposal provide an explicit way to program SIMD-level parallelism while keeping common and trivial parts (e.g. loop iteration transformation) portable. For the next step, we will implement our proposal in the LLVM compiler so that we can try various examples and instruction sets.

References

1. ARM C Language Extensions for SVE. https://developer.arm.com/docs/100987/latest/arm-c-language-extensions-for-sve
2. ARM Scalable Vector Extension. https://developer.arm.com/products/architecture/a-profile/docs
3. DeVito, Z., Hegarty, J., Aiken, A., Hanrahan, P., Vitek, J.: Terra: a multi-stage language for high-performance computing. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, NY, USA, pp. 105–116 (2013). http://doi.acm.org/10.1145/2491956.2462166
4. Estérie, P., Gaunard, M., Falcou, J., Lapresté, J.T., Rozoy, B.: Boost.simd: generic programming for portable simdization. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT 2012, NY, USA, pp. 431–432 (2012). http://doi.acm.org/10.1145/2370816.2370881
5. Ewart, T., Delalondre, F., Schürmann, F.: Cyme: a library maximizing SIMD computation on user-defined containers. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2014. LNCS, vol. 8488, pp. 440–449. Springer, Cham (2014). doi:10.1007/978-3-319-07518-1_29
6. Kretz, M., Lindenstruth, V.: VC: A C++ library for explicit vectorization. Softw. Pract. Exper. **42**(11), 1409–1430 (2012). http://dx.doi.org/10.1002/spe.1149
7. Krzikalla, O., Zitzlsberger, G.: Code vectorization using intel array notation. In: Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing, WPMVP 2016, NY, USA, p. 6:1–6:8 (2016). http://doi.acm.org/10.1145/2870650.2870655
8. Leissa, R., Haffner, I., Hack, S.: Sierra: a SIMD extension for C++. In: Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing, WPMVP 2014, NY, USA, pp. 17–24 (2014). http://doi.acm.org/10.1145/2568058.2568062
9. Petrogalli, F.: A sneak peek into SVE and VLA programming. https://developer.arm.com/hpc/a-sneak-peek-into-sve-and-vla-programming
10. Pharr, M., Mark, W.R.: ispc: a SPMD compiler for high-performance CPU programming. In: 2012 Innovative Parallel Computing (InPar), pp. 1–13, May 2012
11. Pohl, A., Cosenza, B., Mesa, M.A., Chi, C.C., Juurlink, B.: An evaluation of current SIMD programming models for C++. In: Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing, WPMVP 2016, NY, USA, pp. 3:1–3:8 (2016). http://doi.acm.org/10.1145/2870650.2870653