Universität des Saarlandes
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelor's Thesis

Sierra:
A SIMD Extension for C++

submitted by
Immanuel L. Haffner
on **01 April 2015**

Supervisor
Dipl.-Inf. Roland Leißa

Advisors
Dipl.-Inf. Roland Leißa
Prof. Dr. Sebastian Hack

Reviewers
Prof. Dr. Sebastian Hack
Prof. Dr. Jan Reineke

Acknowledgement

I would like to thank my advisor Roland Leißa for offering me this topic as a Bachelor's thesis. Many hours of discussion and hacking with him not only supported me in this venture, but also gave me many useful insights into research and improved my programming skills a lot. His feedback and criticism has helped me profoundly to come up with the results of this thesis.

Thanks also go to my supervisor Sebastian Hack, who is a great teacher and mentor. Without his teachings I would not have been able to complete this task. Furthermore, I want to thank him for reviewing my thesis.

I also want to thank my second reviewer Jan Reineke. His lectures aroused my interest in the program analysis domain and influenced the direction of my further studies.

Abstract

Powerful *Single Instruction, Multiple Data (SIMD)* hardware extensions for data parallel computing are available on many processors. Although they can drastically increase performance of programs, many programs do not utilize these hardware extensions. Targeting SIMD hardware from within general-purpose languages such as C++ compels a highly error-prone, assembly-like programming style. The productivity of the programmer is significantly reduced and portability of the source code is lost. Languages featuring data parallel semantics offer a more convenient and hardware independent way of SIMD programming. Yet, many programmers refuse to redeploy their existing code to another programming language. Automatic approaches are often unable to detect or effectively leverage existing data parallelism, and the programmer can sparsely reason about the compiled program.

We present Sierra: A SIMD Extension for C++. Sierra adds vector types to C++ and overloads the semantics of common C++ constructs in an intuitive way. Code written in Sierra is hardware independent, yet it compiles to specialized SIMD code. The simplicity of Sierra code makes SIMD programs easy to maintain and debug. In contrast to prior approaches, in Sierra the programmer is granted full control over vector lengths.

We evaluated Sierra on a set of public benchmarks. The results show speedups of 2x-4.5x for SSE and 2.5x-7x for AVX. We show the necessary changes to port a program to Sierra. As expected, the process of porting requires only minor changes to the original program.

Contents

Chapter 1

Introduction

With the persistent growth of compute power necessary for research and economy, high-performance computing has become a constant topic of current research. Hardware manufacturers frequently release new processors featuring higher clock frequency, more cores, or more powerful *instruction set architectures (ISA)*.

Because of physical limitations, the rise in clock frequency advances very slowly. For this reason, modern chips feature multiple cores: current CPUs have up to 72 compute cores, while the number of cores on *general purpose GPUs (GPGPU)* ranges into the thousands. Software must be specifically designed to perform efficiently on a multicore architecture. A program using multiple cores is called *task parallel*. The development of task parallel software is a non-trivial and time consuming task. In economy, the cost-benefit-ratio of parallelizing existing sequential code is often very poor. Another approach to improve the efficiency of software is exploiting the very powerful ISA of the concrete hardware. The ISAs of nowadays CPUs usually include SIMD instructions. These SIMD instructions allow to process multiple homogeneous data inside special SIMD registers with only a single instruction; this is referred to as *data parallelism*. SIMD registers come in sizes of 4x32 bits (SSE, AltiVec), 8x32 bits (AVX, AVX2), or 16x32 bits (AVX-512). [1–8]

We call the number of elements that fit into a SIMD register the *vector length*. For SSE we have a vector length of 4 for integers and floats, and a vector length of 2 for doubles.[1] Programs using SIMD instructions can be accelerated by a factor of up to the vector length. As for task parallelism, this is not an easy procedure. Often not only the algorithms of a program, but also the underlying data structures have to be altered in such a way, that data parallelism can be exploited efficiently [9–12].

To make use of languages that feature implicit data parallelism, the performance-critical code must be split off the main program (usually written in general-purpose languages like C++) into a specialized kernel language (e.g. OpenCL [13]). An interface between both languages has to be implemented; a driver has to be launched from the main application to initialize and run the kernel code and to gather the results. Logical

[1]For the remainder of this thesis, we assume integers and floats are 32 bits long, and doubles are 64 bits long.

functionality needed by both the main application and the specific kernel has to be implemented in each language. As porting existing code to a new language is a major effort, many programmers hesitate to adopt such kernel languages.

1.1 Contribution

To allow effective, robust, and human-readable programming of vectorized code in a general purpose language, we present Sierra, a SIMD extension for C++. Sierra extends the C++ language by the new keyword `varying`. Data types annotated with this keyword will be vectorized by the compiler, and operations on these vectorized data types are translated to SIMD instructions. Although the set of available SIMD registers and instructions depends on the ISA and therefore varies from hardware to hardware, Sierra code is independent of the underlying architecture as instruction selection is done during compilation. This way Sierra allows portable programming for SIMD hardware while the source code remains maintainable and reusable. In contrast to languages featuring implicit data parallelism, Sierra gives the programmer the ability to intermix both scalar and vectorized code, and grants him fine-grained control over the vectorization lengths.

This thesis focuses on the code generation for source programs with vector types and vectorized control flow. The underlying concepts of Sierra presented in this thesis can be applied to any programming language and can be seen as a guideline to augment an existing language by vector programming facilities. Vectorization of compound types or pointer types is outside the scope of this thesis and will not be discussed.

1.2 Outline

Chapter 2 gives a brief overview on how Sierra integrates with common C++. Examples demonstrate the use of the `varying` keyword and its semantics. Furthermore, a notion of vectorized control flow is introduced and we explain masking.

In Chapter 3 we present the language `SLang`, a minimal showcase for Sierra, and explain our syntax guided code generation. We take a closer look at the most interesting cases, show optimizations for naïve approaches, and discuss potential pitfalls.

In Chapter 4 we discuss related work, including explicit vectorization at source code level, languages with data parallel semantics, and automatic vectorization techniques.

Our evaluation is presented in Chapter 5. Besides benchmark results we evaluate the process of porting scalar software to Sierra.

Chapter 6 lists future work. We conclude in Chapter 7.

Chapter 2

Sierra at a Glance

2.1 Types

Sierra achieves vectorization of a program by vectorizing the underlying data and converting operations on this data to their vectorial counterparts. Vectorization is triggered by the new type constructor `varying(L)`, which syntactically behaves like a type qualifier in C++. The parameter `L` is a constant expression that specifies the vector length, and must be a positive power of two. The type constructor `uniform` is syntactic sugar for `varying(1)`. We call `uniform` variables *scalar* and `varying(L)` variables *vectorial* (where $L > 1$). Except for uniform types, it is an error to mix vectors with different vector lengths (see Section 2.1.2). In general, a type without a `varying` annotation is scalar. The keyword `uniform` only stresses this fact.

2.1.1 Arithmetic Types

Figure 2.1 demonstrates how the type constructor is used to declare variables of arithmetic vector types. The second line declares `n` as a vectorial `int` variable with vector length 4. The third line declares variable `d` of type vectorial `double` with vector length 4. The size of variable `d` is 256 bits, which may exceed the size of the available SIMD registers. In Section 5.1 we explain how this case is handled internally.

```
int     uniform       u; // scalar int
int     varying(4)    n; // 4-vector of ints
double  varying(4)    d; // 4-vector of doubles
int     varying(7)    m; // error: 7 not a power of 2
int     varying(f())  l; // error: 'f()' not constant
```

Figure 2.1: *Declaration of vectorized arithmetic types*

Arithmetic Conversions. The C++ Language Specification defines specific rules when and how values are automatically converted from one type to another. In Sierra these rules apply analogously to vectors:

```
short varying(4) s;
int   varying(4) i;
s + i; // s is converted to int varying(4)
```

2.1.2 Broadcast

Although in general intermixing vector types of different lengths is forbidden, intermixing uniform and vectorial data is allowed:

```
int uniform    u = 3;
int varying(L) v = u;
int varying(L) w = u + v;
```

In the assignment of the second line, the value of the uniform variable `u` is *broadcast* to the vector length of the left-hand side. This yields a new vector of length `L`, where each element has value 3. This vector is then assigned to `v`. Similarly the value of `u` is broadcast in the addition on line 3.

2.1.3 Initializer List

In C++, variables can be initialized on declaration. In Sierra vectorial variables can be assigned or initialized with a scalar value through broadcasting (see Section 2.1.2). Alternatively a vectorial variable can be initialized by specifying an *initializer list* of length equal to the variable's vector length. Syntactically, an initializer list behaves like a structure initializer.

```
int  varying(4) v = {0, 1, 2, 3};
```

Here, the first element of `a` will be assigned 0, the second element will be assigned 1, and so on. Additionally, Sierra implements the concept of compound literals, introduced with C99, for vectors:

```
int varying(4) v;
v = (int varying(4)) {0, 1, 2, 3};
```

2.1.4 Element Access

To access and manipulate specific elements of a vector we introduce the following two functions:

```
template<class T, int L>
T varying(L) insert(T varying(L) &vector, int i, T value) {
    ...
}

template<class T, int L>
T extract(T varying(L) vector, int i) {
    ...
}
```

The function `insert` assigns `value` to the vector's element at index `i`; the function `extract` returns the value of the vector's $\texttt{i}^{th}$ element. Note that both function templates are parametric in the vector length.

It is important to understand that these functions must be provided by Sierra and must not be defined by the programmer. As the internal representation of a vector varies with the target machine's ISA, accessing an element of a vector via aliasing yields undefined behaviour.

2.2 SIMD Mode

Sierra leaves the semantics of the programming language unchanged as long as operations are performed solely on values of scalar type. As soon as a Sierra vector type is involved in a computation or influences control flow, the program enters Sierra's *SIMD Mode*. Inside SIMD Mode, the semantics of the language is extended to vector types. Control flow is vectorized and the semantics of control flow constructs is adopted accordingly. In vectorized control flow, operations are performed on so called *SIMD Lanes*, which are indexed from 0 to $n-1$, where n is the vector length of the vector type that triggered the SIMD Mode. An instruction executed in SIMD Mode is executed on each SIMD Lane simultaneously (see Section 2.2.2). We distinguish between *live* and *dead* lanes. Dead lanes do not execute instructions, leaving original data unmodified. SIMD Lanes may become live or dead during program execution (see Section 2.2.1).

In scalar control flow, logical control flow divergence can be implemented with conditional branches. In vectorized control flow, a conditional branch may skip live lanes, harming the program semantics. Therefore control flow is converted to data flow. Conditional branches are removed to *flatten* the control flow and operations are *masked* [12, 14, 15].

2.2.1 Masking

SIMD instructions are masked to target specific elements of a vector. Masking effectively selects which lanes are live and which ones are dead. The example in Figure 2.2 demonstrates masking of operations inside a vectorial `if`-statement. The value 3 is broadcast and compared to the variable `v`, evaluating to the vector `{true, false, false, true}`

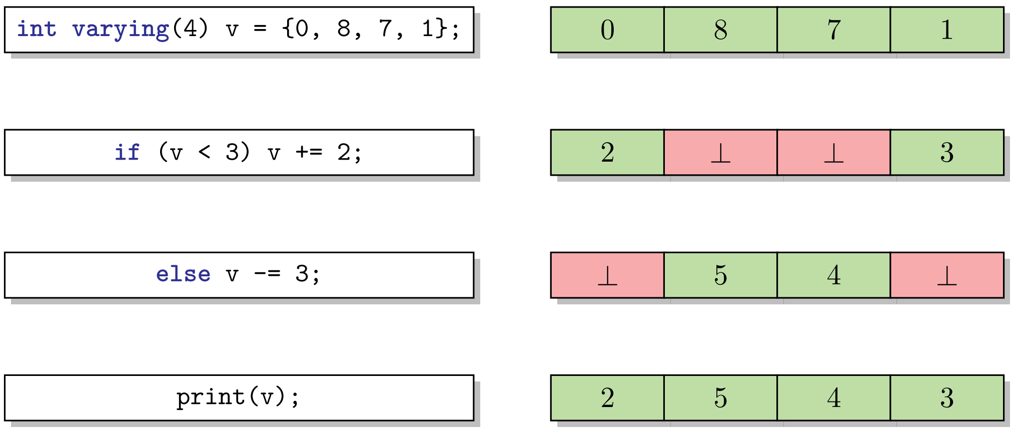


Figure 2.2: *Automatic masking in vectorized control flow*

of type `bool varying(4)`. We call this vector the *(current) mask*.[1] The current mask selects which lanes are live in the *true*-successor, here the lanes with index 0 and 3. The negated current mask selects which lanes are live in the *false*-successor, i.e. lanes 1 and 2. The symbol $\perp$ denotes that the value of the element is undefined at this point in the program. When control reaches the `print(v);` statement, the live lanes of the *true*- and *false*-successor are joined again.

Code Generation

Depending on the available hardware, Sierra uses different techniques to realize the masking. The examples in Figure 2.3 show two implementations of the example from Figure 2.2.

In Figure 2.3a the underlying instruction set is AVX-512. This instruction set allows to *blend* instructions, meaning that instructions take an additional mask argument (predicate), and computations are only performed on the live lanes. This technique is called *predicated execution*.

Figure 2.3b shows an implementation using the SSE instruction set, which does not feature implicit masking. Therefore, we have to produce new vectors for both the *true*- and the *false*-successor, and merge them into the result vector using the mask.

In comparison, AVX-512 has lower register pressure since no copies have to be made before modifying data. Furthermore, AVX-512 needs less instructions, since explicit masking can be omitted.

[1]We will see later on that during program execution multiple masks are used. The current mask always refers to the mask used to determine which lanes are live/dead at the current point of execution. Each program point has at most one current mask.

```
SIMD_Load( v, {0, 8, 7, 1} );
SIMD_Lt( mask, v, {3, 3, 3, 3} );
SIMD_BlendAdd( mask, v, {2, 2, 2, 2} );
SIMD_Neg( mask, mask );
SIMD_BlendSub( mask, v, {3, 3, 3, 3} );
```

(a) *Predicated execution using AVX-512 blend instructions.*

```
SIMD_Load( v, {0, 8, 7, 1} );
SIMD_Lt( mask, v, {3, 3, 3, 3} );
SIMD_Add( then, v, {2, 2, 2, 2} );
SIMD_Sub( else, v, {3, 3, 3, 3} );
SIMD_Select( v, mask, then, else );
```

(b) *Explicit masking with SSE instructions.*

Figure 2.3: *Assembly-like pseudo code for Figure 2.2.*

2.2.2 Lock-Step Semantics

The C Language Specification defines sequenced execution as *"The presence of a sequence point between the evaluation of expressions A and B implies that every value computation and side effect associated with A is sequenced before every value computation and side effect associated with B."* [16] We extend the definition of sequenced execution to vector operations. When a sequence point is reached, all operations before the sequence point are guaranteed to have terminated execution *on all SIMD lanes.* The sequence points in Sierra constructs are placed at the same positions as in their scalar counterparts. However, to allow various implementations and optimizations, we leave the order in which elements of a vector are processed unspecified.

Chapter 3

Code Generation

In this chapter we will discuss Sierra's code generation. As generating scalar SSA form is a well-known procedure [17], we will mainly focus on code generation for vector constructs, i.e. when inside SIMD Mode. Section 3.1 presents the small, C-like programming language `SLang`. In Section 3.2 we show how to translate `SLang` into an SSA-form *intermediate representation (IR).* Section 3.3 explains how control flow divergence is based on vectorial conditions. Section 3.4, Section 3.5 and Section 3.6 show how the implicit masking embedded in the semantics of `SLang` is implemented by explicit masking in the IR. Last but not least Section 3.7 presents a technique do implement short-circuit evaluation on vector expressions.

3.1 The Language `SLang`

Figure 3.1 describes the syntax of `SLang`, a subset of standard C++ augmented by the type constructor `varying`, designed to be a minimal showcase for Sierra. An expression is a constant[1], a vector of constants where the elements are separated by commas, a variable, a logically negated expression, a binary expression with an arithmetic, relational, or assignment operator, or a function call with arbitrarily many arguments, separated from each other by commas. A type is an unqualified type, i.e. bool or int, optionally qualified by varying(L), where L is a constant. The rules for type conversion have already been stated in Section 2.1. The Scope statement declares a new scope that begins with the opening braces "{" and ends with the closing braces "}". The inner part $\overline{s}$ expands to a (potentially empty) list of statements. Declarations inside a scope are only visible from within the same scope. Since `SLang` has no Void type, functions always return a value; the Return statement always requires an expression. Functions can be defined with arbitrarily many parameters, separated from each other by commas. Functions must be defined on declaration, forward declarations are not possible.

As already mentioned, pointer and compound types are outside the scope of this thesis, and therefore have been omitted in `SLang`.

[1]We limit constants to numeric literals.

e ::=	*Expression*	s ::=	*Statement*
c	Constant	$\{\overline{s}\}$	Scope
$\mid \{\overline{c}\}$	Vector Constant	$\mid e;$	Expression-Statement
$\mid$ id	Variable	$\mid t$ id;	Declaration
$\mid\ !\ e$	Not	$\mid$ if (e) s else s	If-Else
$\mid e$ op e	Binary Operation	$\mid$ while (e) s	While
$\mid$ id$(\overline{e})$	Function Call	$\mid$ break;	Break
		$\mid$ continue;	Continue
t' ::=	*Unqualified Type*	$\mid$ return e;	Return
bool			
$\mid$ int		f ::= t id$(\overline{t\ \text{id}})$ $\{\overline{s}\}$	Function
t ::=	*Type*		
$\mid t'$			
$\mid t'$ varying(L)	Vector Type		

Figure 3.1: *Syntax of* `SLang`

3.2 SSA Construction for Statements

The pattern in Figure 3.2 shows how vectorial statements are translated to SSA form. Rectangular boxes represent basic blocks, while clouds represent arbitrarily complex sub-graphs. Every statement has an incoming mask and an outgoing mask. We create a single entry and a single exit for each statement, this makes it simpler to establish dominance. The incoming mask must dominate the body of a statement and the outgoing mask must post-dominate the body. The incoming mask initializes the statement's current mask. All instructions inside a statement are masked with the current mask. The current mask can change during execution of the statement. The outgoing mask of a statement is used as the incoming mask for its successor. Scalar statements do not have any incoming or outgoing mask.

3.2.1 From Scalar to Vectorial

When code generation enters SIMD Mode, there is a special case for the incoming mask of a statement. The original context is scalar and the statement for which code is to be generated is vectorial. Hence, the predecessor statement is scalar, and as such has no outgoing mask. The examples from Figure 3.3 demonstrate two such scenarios.

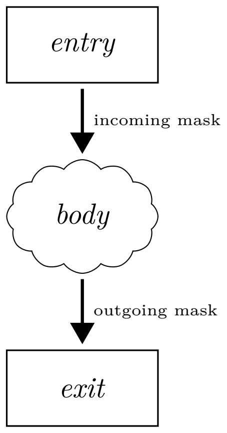


Figure 3.2: *General form of a vectorial statement in SSA form*

```
1 bool varying(4) cond;
2 ...
3 if (cond) {
4     // SIMD Mode
5 }
```

(a)

```
1 int i;
2 int varying(4) j;
3 ...
4 i + j;
```

(b)

Figure 3.3: *Triggering SIMD Mode*

In Figure 3.3a we see an If statement with the condition `cond`. Since `cond` is vectorial, it renders the If statement vectorial, too. Hence, code generation for this statement expects an incoming mask, yet there is no current mask at this point. We will create a new mask with the same vector length as `cond` and set each lane live. This mask is the If statement's current mask.

In line 4 in the example in Figure 3.3b we see the expression `i + j`. Because the variable `j` is vectorial, so is the expression, and hence the expression statement. Since the context is scalar, code generation must enter SIMD Mode before code for the expression can be generated. As in the previous example, a mask with vector length 4 is created and all lanes are set live. This mask is the current mask for the Expression statement.

3.2.2 From Vectorial to Scalar

In general, code generation exits SIMD Mode after finishing code generation for the statement that triggered the SIMD Mode. In case of Figure 3.3a, the SIMD Mode is triggered at line 3 and exited at line 5.

However, there is a special case where code generation must remain in SIMD Mode. In Figure 3.4 the If statement triggers SIMD Mode, and the expression `return 42` becomes vectorial. For the lanes 0 and 3 the function returns 42, and these lanes are rendered dead for the remainder of this function. Program execution continues with the successor

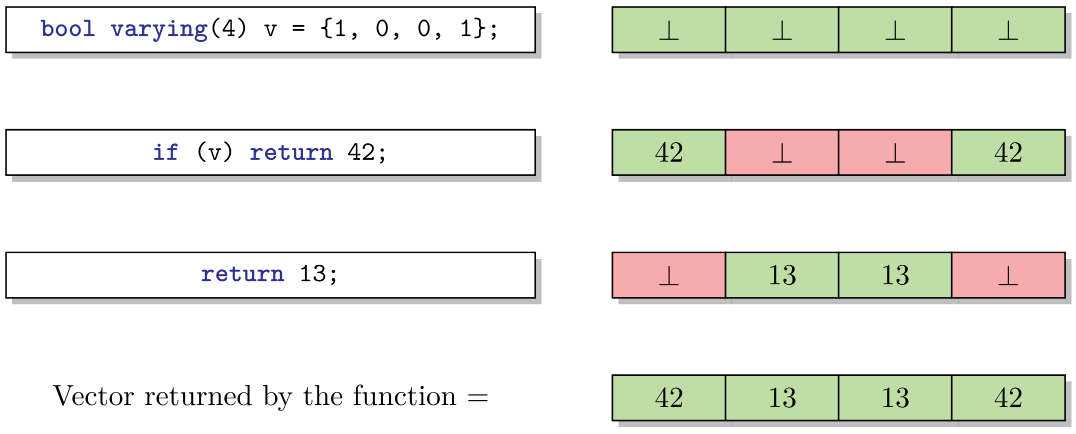

bool varying(4) v = {1, 0, 0, 1};
⊥ ⊥ ⊥ ⊥
if (v) return 42;
42 ⊥ ⊥ 42
return 13;
⊥ 13 13 ⊥
Vector returned by the function =
42 13 13 42

Figure 3.4: *Vectorized* `return`

of the If statement. When the expression `return 13` is reached, it must be executed only on lanes 1 and 2. Hence, code for the expression must be generated from within SIMD Mode.

Whenever code generation encounters a `return` inside SIMD Mode, the outgoing mask of the statement that triggered SIMD Mode is used as mask for all successors of that statement, even if the context was scalar. Here, the outgoing mask of the If statement is `{0, 1, 1, 0}` and is used as current mask for the second `return`-statement.

3.2.3 Nested Statements

In `SLang` there are only three statements that produce nesting. A statement nested inside a Scope statement simply uses the mask of its predecessor as incoming mask. The first statement in a Scope uses the Scope's incoming mask. A statement nested in a While statement or the *then*-statement of an If statement uses the evaluated condition as mask. A statement nested in the *else*-statement of an If statement uses the negated condition as mask.

All computations performed to evaluate the condition are masked with the statement's current mask. Hence, only already live lanes may be live in the nested statement; dead lanes remain dead.

Figure 3.5 demonstrates masking of nested statements. The body of the While statement uses the evaluated expression `v > 2` as incoming mask, which is `{0, 0, 1, 1}` in the first iteration. The If statement uses this mask as incoming mask. It evaluates the condition with respect to its current mask. This condition is used to mask the *then*- and the *else*-statement of the If. This means, no matter what the value of `v` is, the lanes 0 and 1 will remain dead in the *then*- and the *else*-statement. However, the value of `v` is used to mask the remaining live lanes.

We see that during code generation in SIMD Mode, multiple masks may be used. In the example we create an incoming mask for the While statement to enter the SIMD Mode. Then there is an incoming mask for the body of the While statement, which is

```
int varying(4) v;
v = {0, 1, 2, 3};
while (v > 2) {     // current mask = {0, 0, 1, 1}
    if (v % 2) {
        // current mask = {0, 0, 0, 1}
    } else {
        // current mask = {0, 0, 1, 0}
    }
}
```

Figure 3.5: *Nested statements in* SLang

computed at runtime by evaluating the While's condition. Afterwards the *then*- and the *else*-statement have an incoming mask each. It is possible that multiple masks reach the same program point, but only one of them, i.e. the current mask, is used to mask instructions and determine which lanes are live or dead. Because the computation `v % 2` is already masked with `{0, 0, 1, 1}`, it evaluates to `{0, 0, 0, 1}`, which is the incoming mask of the *then*-statement. Negating this mask to compute the incoming mask of the *else*-statement yields `{0, 0, 1, 0}`, as the negation is also masked.

3.3 Branching on Vectorial Conditions

In previous examples we already saw If and While statements with vectorial conditions. In a scalar setting we would translate these to code with conditional branches. SIMD allows us to process multiple homogeneous data at once with a single instruction, but not to execute multiple instructions at once. Hence we cannot execute both sides of a conditional branch simultaneously. Instead we have to schedule execution of one side before the other. Section 3.5 and Section 3.6 will explain in detail how this scheduling is performed by the code generation.

We however need conditional branches that depend on vectorial conditions, e.g. for While statements. We will see later on, that a While loop iterates as long as the condition evaluates to a vector with at least one element being not 0. We introduce the following tests that take a bool vector as argument and return a scalar bool. These tests will be inserted by the code generation wherever necessary. The concrete implementation of these tests depends on the available ISA.

All-True. Evaluates to 1 if the tested bool vector is true on *all* currently live lanes, 0 otherwise.

Any-True. Evaluates to 1 if the tested bool vector is true on *any* currently live lane, 0 otherwise.

3.4 Vectorial Functions

A function is vectorial if it returns a vector. Vectorial functions can be invoked from both scalar and vectorial contexts. In case the context is scalar, code generation enters SIMD Mode as already stated in Section 3.2.1. It is forbidden to call a vectorial function from within SIMD Mode if the vector lengths of the returned value and the current mask differ. The incoming mask of the Call expression is passed as hidden parameter to the function and used as the incoming mask of the first statement in the function body.

In Section 3.2.2 we already explained how a Return statement influences masking. As multiple Return statements may be executed within a single function call, the returned value must be stored until control flow returns to the caller. To do so, every function keeps a return vector of the return type. Each Return statement stores the value of the evaluated expression to this vector. Of course, the store is only executed on the currently live lanes. The right side of Figure 3.4 shows the value of the return vector at different points of execution.[2] If execution of the function does not diverge, each element of the vector was assigned a value exactly once. When a return was executed on every lane that was live at the entry of the function body, the function returns to the caller, with the assembled return vector as return value.

3.5 Vectorial `If`

A scalar If statement has a scalar condition. This condition is evaluated and converted to bool. In case the value is 1 control flow continues in the *then*-statement, and in the *else*-statement otherwise. After the then- or the else-statement has been processed, program execution continues at the successor of the If statement. As the condition selects which statement is executed, control flow diverges.

In the vectorial setting, the condition of the If statement is vectorial. Hence, we cannot simply decide whether to execute either the then- or the else-statement. Instead, both statements are executed and the condition is used to mask the instructions inside the statements. Figure 3.6 shows shows how a vectorial If statement is translated to SSA form. Let us initially ignore the dashed arrows. We see that there is no control flow divergence. In the *if-entry* block, the vectorial condition is evaluated to the condition mask. Then program execution continues in the then-statement with the condition mask as incoming mask. After the then-statement has been processed, the else-statement is executed with the incoming mask being the negated condition mask. Program execution continues in *if-exit*, where the live lanes of the outgoing masks of both statements are joined, forming the outgoing mask of the If statement.

The dashed arrows show optional branches that may improve program performance. If every element of the condition mask is 0, execution of the then-statement becomes superfluous. The same holds for the else-statement if every element of the condition mask is 1. Therefore we add an *Any-True* check in *if-entry* to decide at runtime whether

[2] At the second Return statement the value of the return vector is $\perp$ on lanes 0 and 3 because the corresponding lanes are dead.

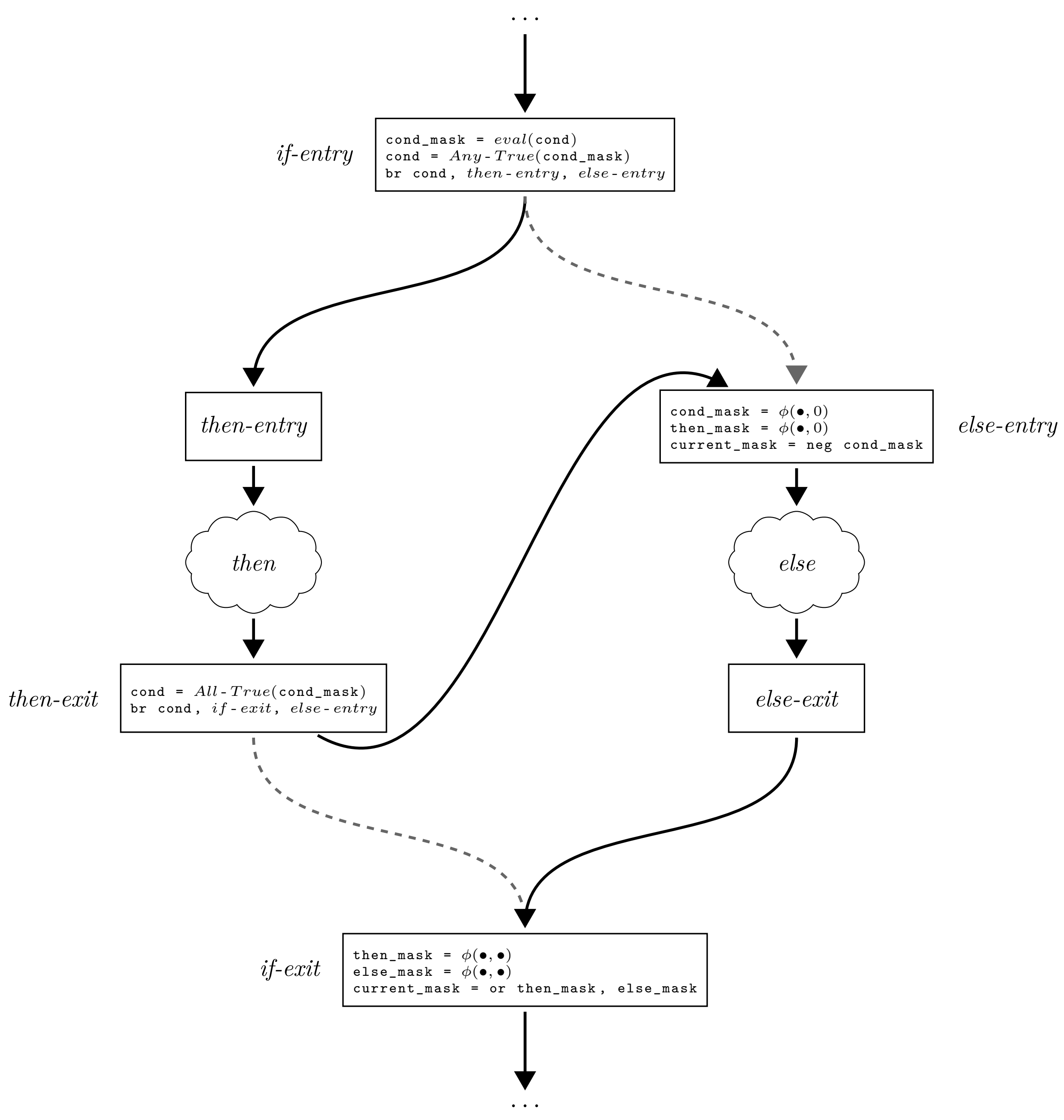

...
if-entry
cond_mask = eval(cond)
cond = Any-True(cond_mask)
br cond, then-entry, else-entry
then-entry
then
then-exit
cond = All-True(cond_mask)
br cond, if-exit, else-entry
cond_mask = φ(●, 0)
then_mask = φ(●, 0)
current_mask = neg cond_mask
else-entry
else
else-exit
if-exit
then_mask = φ(●, ●)
else_mask = φ(●, ●)
current_mask = or then_mask, else_mask
...

Figure 3.6: *If statement translated to SSA form*

to skip execution of the then-statement and immediately proceed with the else-statement. Similarly, we introduce an *All-True* test in *then-exit*.

By adding these conditional branches we implemented the dashed arrows. The translated If statement now contains control flow divergence. This introduces a new problem: the outgoing mask of the *then*- and the *else*-statement no longer dominate the *if-exit* block. We introduce two ϕ-functions in *if-exit*, one for the outgoing mask of *then* and *else* each. Note that the outgoing mask of the *then*- respectively the *else*-statement is not necessarily equal to the incoming mask. We need to introduce two more ϕ-function in *else-entry*: one for the condition mask and one for the outgoing mask of the *then*-statement.

3.6 Vectorial `While`

The SSA form of a vectorial While loop looks similar to its scalar counterpart. The *while-head* block has two incoming edges, one from the predecessor statement and one from the exit of the loop body. Inside the loop header the vectorial condition is evaluated to the condition mask. An *Any-True* check of the condition mask is performed, and if it evaluates to 1 program execution continues in the loop body. Otherwise, loop iteration terminates and execution continues in *while-exit*. Code for the loop body is generated and the condition mask is used as its incoming mask. The outgoing mask of the loop body is then passed back to the loop header, where it is used as current mask. As the loop header has two predecessors, we introduce a ϕ-function to select the incoming mask.

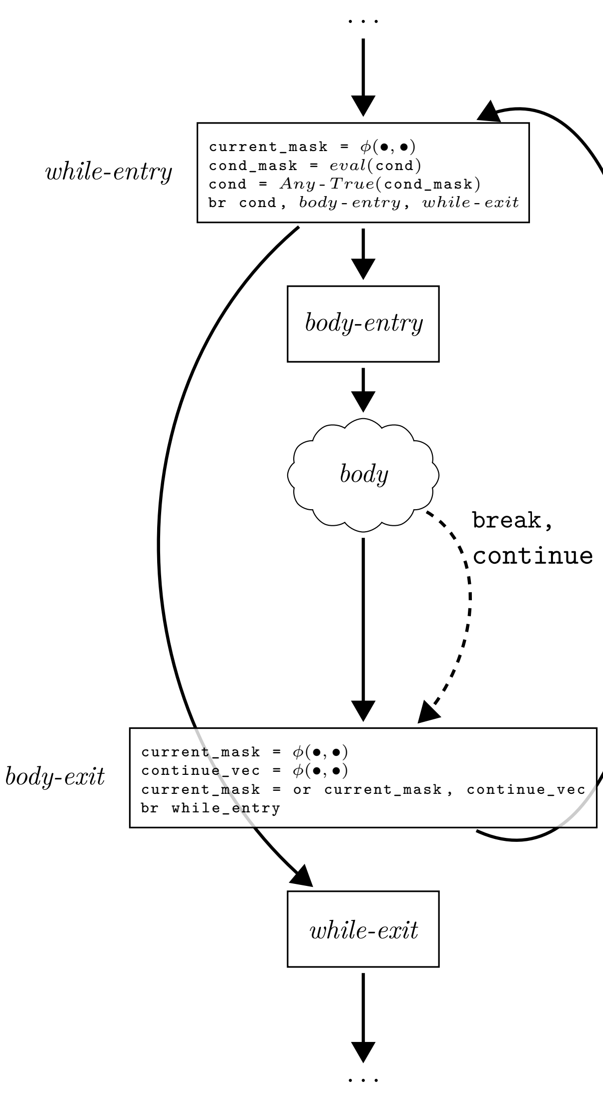


Figure 3.7: *While statement translated to SSA form*

Break and Continue

When a `break` is encountered, all currently live lanes are set dead for the remainder of the whole loop. Again, we can add an additional check here to improve program performance: if the current mask is *All-True* with respect to the current mask of the loop body, the `break` can directly jump to the loop's exit block.

A `continue` renders all currently live lanes dead, but only for the remainder of the current loop iteration. When *body-exit* is reached, all lanes that were set dead by a Continue statement are set live again. Therefore, every loop keeps a continue vector, a bool vector with the same vector length as the current mask. Initially, all elements are set to 0. When a `continue` is executed, it sets the elements on the currently live lanes

```
 1 int varying(4) v;
 2 v = {0, 1, 2, 3};
 3
 4 while (v < 5) {
 5     if (v % 3)
 6         break;
 7     if (v == 0) {
 8         v = 2;
 9         continue;
10     }
11     v = v * 2;
12 }
```

Figure 3.8: *Demonstration of Break and Continue*

of the continue vector to 1. In the *body-exit* block, the continue vector is used to set the lanes, that were rendered dead by a Continue statement, live again. Additionally, we introduce an *All-True* test with respect to the loop body's current mask for each Continue statement to directly branch to *body-exit* if execution of the remainder of the loop body becomes superfluous.

The example in Figure 3.8 shows a loop with both a Break and a Continue statement. In the first iteration, the condition `v < 5` evaluates to `{1, 1, 1, 1}`. Then `v % 3` is evaluated to `{0, 1, 1, 0}`. The `break` renders lanes 1 and 2 dead for the remainder of the loop. Execution continues in the If statement on line 7. The expression `v == 0` evaluates to `{1, 0, 0, 0}`. After the assignment `v = 2`, the Continue statement renders lane 0 dead for the remainder of this iteration. The assignment `v = v * 2` is then executed with the current mask `{0, 0, 0, 1}`. The value of variable `v` after the first iteration is `{2, 1, 2, 6}`. Before the program returns to the loop header, lane 0 is set live again. The outgoing mask of the loop body, i.e. `{1, 0, 0, 1}`, is then used as incoming mask for the loop header.

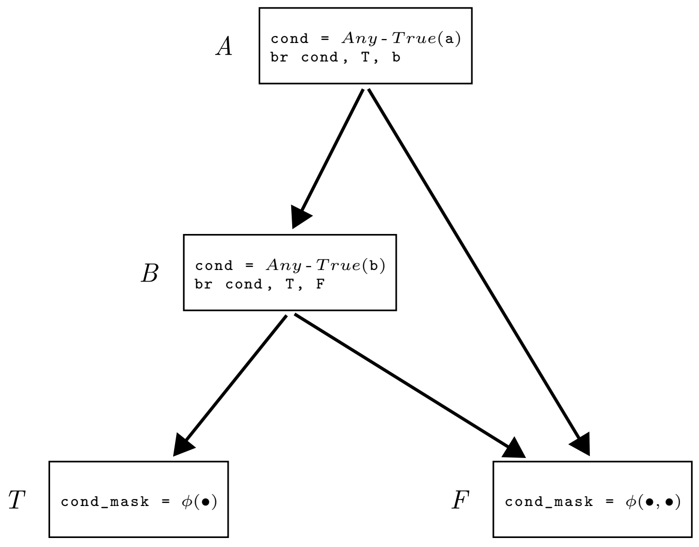
In the next iteration, the condition evaluates to `{1, 0, 0, 0}`, as lanes 1 and 2 remain dead. The last live lane is set dead by the `break` in line 6. As all lanes are dead and the continue vector is *All-False*, program execution continues at the loop exit.

3.7 Vectorial Short-Circuit Evaluation

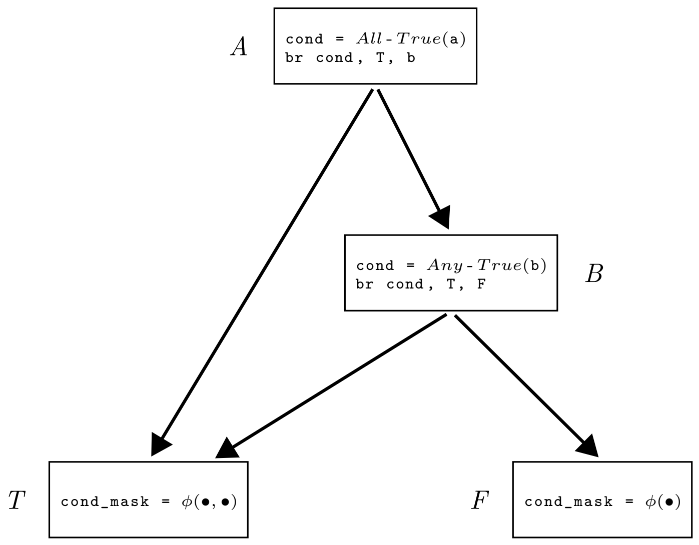
The short-circuit operators `&&` and `||` evaluate their right-hand side only if necessary. In the scalar setting, the expression `a && b` would only evaluate `b` if `a` evaluated to 1.

Sierra tries to mimic this behavior when evaluating vectors. Figure 3.9a shows a logical and (`a && b`) translated to SSA form, where `a` and `b` are vectors. If `a` is *All-False*, execution of the T-successor is superfluous, and `b` must not be evaluated. Hence, the conditional branch directly jumps to F and B is not executed. Otherwise, `b` is evaluated. If `b` is *Any-True*, T is executed, F otherwise.

Figure 3.9b shows a similar graph for the translated vectorial logical or (`a || b`). This time execution of F becomes superfluous if `a` is *All-True*. If this is not the case, B is executed. If *Any-True* of `b`, T is executed, F otherwise.



(a) `a && b`



(b) `a || b`

Figure 3.9: *Short-Circuit Evaluation for logical and/or*

The graphs do not show a control flow edge from T to F. Inserting such an edge is not job of the code generation for expressions, but has to be done by the code generation for statements such as If and While.

Let us have a look at a more complex example of vector short-circuit evaluation. Figure 3.10 shows the SSA graph for evaluating the expression `(a && b) || (c && d)`. Code generation starts at the top of the expression tree, the operator `||`. It recursively triggers code generation for its left-hand and right-hand side. First, the left-hand side is processed. The operator `||` selects the original T as block T and C as block F for the pattern in Figure 3.9a, and build the corresponding graph. So far, we constructed the edges $A \rightarrow B$, $A \rightarrow C$, $B \rightarrow T$ and $B \rightarrow C$ Next, code for the right-hand side of `||` is emitted. Again, we simply instantiate the pattern from Figure 3.9a, this time with both T and F in the pattern being T respectively F from the original graph. This adds the edges $C \rightarrow D$, $C \rightarrow F$, $D \rightarrow T$, and $D \rightarrow F$. Code generation for the expression is done and we obtain the graph from Figure 3.10.

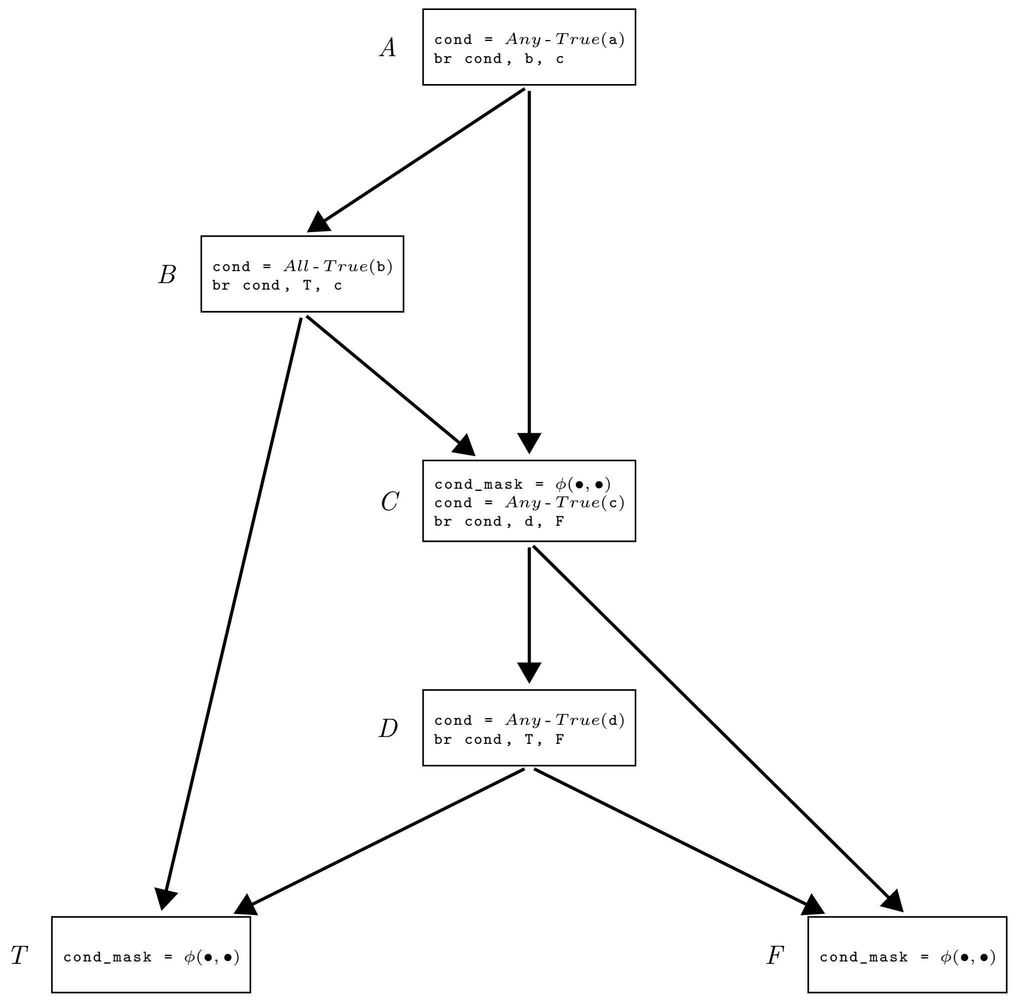


Figure 3.10: *Short-circuit evaluation of* `(a && b) || (c && d)`

In the following example, sub-expression `b` will not be evaluated, as `a` already satisfies the expression `a || b` on each lane. Similarly, sub-expression `d` is not evaluated because of `c`.

```
a = {1, 1, 1, 1};
c = {0, 0, 0, 0};
...;
(a || b) && (c && d);
```

Chapter 4

Related Work

4.1 Intrinsics and Boost.SIMD

Compilers often cannot harness the full power of the underlying hardware due to restrictions to the static analysis and optimization framework. Therefore, compiler-specific language extensions to embed assembly code directly into the higher-level language and intrinsic functions have been developed. These allow programmers to tune their program on a very fine-grained level for specific architectures. However, this encourages an assembly-like, highly error-prone programming style. The source code is obfuscated by hardware-specific instructions, not only hiding the program's logic and impeding debugging, but also making the code non-portable.

In the special case of vectorized programming, the vectorized control-flow must be implemented by the programmer himself. This means that not only the masks have to be computed, but also results of operations have to be masked manually (see Figure 2.3).

4.1.1 Boost.SIMD

To overcome the hardware dependence of assembly code and to offer an abstraction to vectorized programming, the C++ `Boost.SIMD` library [10] has been developed. It is designed as an embedded domain specific language for C++ using `Boost.Proto`.

`Boost.SIMD` introduces the container class `pack<class T, int N>`, where `T` must be of arithmetic type and `N` must be a power of 2, as a new layer of abstraction. A concrete implementation for `pack` will be determined by `Boost.SIMD` at compile time, such that special SIMD registers will be used to represent `pack`. The class `pack` internally takes care of memory fetching strategies, such as vectorial load/store and gather/scatter.

To offer effective computation, operators and functions have been overloaded or reimplemented for the `pack` class. It is also possible to evaluate a `pack` as a boolean expression or compare `pack`s of same length, yielding a new `pack` of the same length of type `logical`.

Explicit masking is available through the use of the special `if_else` function.

```
x = if_else(x != 0, x * y, y);
```

The function takes the first argument, evaluates it to `pack<logical<T>>` and uses it as a mask to assemble a new `pack<T>` from the second and third argument and returns. Before the function is called, the arguments must be evaluated. Thus, the evaluation of `x * y` happens before the function call and is independent of the condition.

In contrast to Sierra, `Boost.SIMD` does not feature implicit vectorized control flow. Therefore, masking has to be done explicitly using the `if_else` function, which directly translates to code similar to Figure 2.3b. The programmer is forced to implement vectorized control flow himself, making extensive use of the `if_else` function. In Sierra, vectorized control flow is already embedded into the language semantics. The compiler will generate the necessary masking code, no programmer intervention is necessary. The container `pack` allows to specify the vector length manually. If the proposed vector length does not fit into a single SIMD register, the data is distributed among multiple registers. This is similar to Sierra's double-pumping.

4.2 Array notation

Languages such as APL [18] and VectorPascal [19] have been developed to express data-parallelism in scalar code. Existing languages, including C++ [20], were extended by array notations to aid exposing data-parallelism to the compiler. Operations performing on whole arrays (or sections) that make use of these array notations can be vectorized by the compiler if profitable.

4.2.1 Intel® Cilk™ Plus

Cilk Plus [20, 21] is an extension for C/C++ to address both multicore and vector processing. The core feature is the new operator `[:]`. Expressions of the form `array-expression[lower-bound : length : stride]` can be used to delineate an array section of the array pointed to by `array-expression` with a size of `length` many elements. The stride is used to only select every n^{th} element, defaults to 1.

The semantics of C/C++ are extended in such a way, that operations such as arithmetic operations or function calls can be performed directly on array-sections. The following example demonstrates the use of the new Cilk Plus array notation.

```
if (0 == a[:])
    result[:] = true;
foo(result[:]);
```

In the first line, all elements of the array `a` are compared to zero. For every index `i`, where `0 == a[i]`, the second line will be executed, such that `result[i] = true`. Afterwards, the function `foo` is called with the array-section `result[:]` as argument. If `foo` is declared only for scalar arguments, the function call will simply be performed on every array element. The function could also be declared as a *SIMD-enabled function* by the annotation `__declspec(vector)`, telling the compiler to generate both a scalar version of the function and a vector version. SIMD-enabled functions can perform on both scalar data and array-sections.

Array-sections can be used as indices to access arrays, resulting in gather/scatter operations:

```
C[:] = A[B[:]]  // gather
A[B[:]] = C[:]  // scatter
```

The semantics of the language, and especially the `[:]`-operator, allow the compiler to perform vectorization during compilation. Array-sections are split among SIMD registers, and operations on array sections are translated to SIMD instructions. Whole functions can be vectorized by passing array-sections as arguments. Again, the programmer has no influence on vectorization lengths and vectorization granularity.

4.3 Single Program, Multiple Data

A different approach to exploiting SIMD hardware is providing a new language, that expresses (data-) parallelism, coupled with a compiler, mapping this language-parallelism to concrete hardware. In the Single Program, Multiple Data (SPMD) approach, the programmer writes his program in a language that appears to be scalar, but features parallel semantics. The compiler creates multiple instances of the same, scalar program. These versions can then be mapped to SIMD lanes or processing cores.

With this approach, the programmer does not have to struggle with vectorization details, such as register allocation, operator overloading, or masking. Besides that, portability of the code relies on the compiler.

4.3.1 Intel® SPMD Program Compiler

The Intel SPMD Program Compiler (ISPC) [12] is a compiler for a new, C-based language extended by SPMD constructs. ISPC supports lightweight function calls from C++ to ISPC and vice versa, and guarantees coherent shared memory between both languages. Together with the syntactical similarity to C, this encourages the programmer to redeploy computationally intensive kernel code to ISPC.

The example from Figure 4.1 shows the definition of an ISPC kernel. The `export` keyword is used to make the function callable from C++. The `uniform` keyword behaves as in Sierra, defining data as scalar. The declaration of the induction variable `i` is not preceded by `uniform`. Therefore, the ISPC compiler will produce one instance of `i` for every program instance. The variable is initialised with `programIndex`, which is a built-in variable that assigns each program instance an index from zero to `programCount`. The

```
export void update(uniform float a[],
                   uniform float b[].
                   uniform int n) {
    for (int i = programIndex; i < n; i += programCount) {
        if (b[i] > a[i])
            b[i] -= a[i];
    }
}
```

Figure 4.1: *An ISPC kernel*

built-in variable `programCount`, which holds the number of created program instances, is then used to increment the induction variable.

For this example, the compiler can create vector length many program instances and distribute them to SIMD lanes. The result is a vectorized loop with a vectorized induction variable of length `programCount`.

Making use of ISPC enforces the programmer to split his program among two languages. Computationally intensive kernel code, which is implemented in ISPC, must be compiled separately using a special ISPC compiler. The semantics of ISPC restrain the compiler to use only a single vector length per translation unit for vectorization. On the language level, the programmer has no influence on the vectorization lengths, and therefore intermixing code of different vector lengths is not possible. In contrast to ISPC, in Sierra the program starts in a scalar context and vectorization is triggered manually by the programmer by making use of vector types.

4.3.2 Intel® Array Building Blocks

Intel's Array Building Blocks (ArBB) [22, 23] is an approach similar to ISPC, targeting both SIMD and multi-core architectures. It evolved from the former Intel research project Ct (*C for Throughput Computing*) and RapidMind, a multi-core development platform for both Intel and AMD hardware.

ArBB provides the programmer with a C++-like language for kernel programming, augmented with an expressive set of types, including dense, indexed, and sparse containers. At compile time, the source code is translated into an intermediate representation. At runtime, a JIT compiler optimises the IR for the detected target architecture. The binary is executed inside the ArBB Virtual Machine, which consists of the Threading Runtime (TRT), the Memory Manager (MM), and the Heterogeneous Runtime (HRT). TRT takes care of implicit synchronization, MM and TRT partition data for parallel processing. HRT coordinates loading and executing code and communication with acceleration devices (e.g. GPGPUs).

In contrast to ISPC, the programmer is forced to produce boilerplate code to format the scalar data before the ArBB kernel can be invoked. On the first execution of a ArBB kernel the JIT compiler is invoked; re-execution of the same code reuses the cached compilation results.

4.3.3 OpenCL

The Open Computing Language (OpenCL) heterogeneous computing on a variety of modern CPUs, GPUs, and other microprocessor designs. OpenCL is an open standard maintained by the non-profit technology consortium *Khronos Group*. OpenCL 1.0 was published in 2008, the latest version at present is OpenCL 2.0, released in 2013.

As in ISPC and ArBB, OpenCL forces the programmer to split the implementation among two languages. OpenCL kernels are implemented in the OpenCL C language, a derivate of C99. OpenCL C is enriched by OpenCL keywords and special vector types, and omits function pointers and recursion. Similar to ISPC, OpenCL C features SPMD semantics. The built-in function `get_global_id()` returns an integer unique for each program instance. (It is similar to ISPC's `programIndex`.) Implicit vectorization and parallelization happen during the compilation process.

The host language has to do a set of routines to facilitate an OpenCL kernel. An OpenCL *compute context* must be created to address the target hardware. A *command queue* must be created to enable communication with the target device. *Buffer memory objects* must be allocated to pass data to and from the target device. The OpenCL program must be compiled, which usually happens at runtime. The kernel arguments must be set, passing the buffer memory objects as I/O memory. Finally, the execution of the kernel is enqueued on the target device.

4.4 OpenMP

Open Multi-Processing (OpenMP) [24] is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs. The latest version, OpenMP 4.0, introduces preprocessor annotations to address SIMD hardware. The SIMD construct `#pragma omp simd` is used to indicate that a loop can be transformed into a SIMD loop, where multiple iterations of the loop are executed concurrently using SIMD instructions. (This annotation is similar to the ICC/GCC extension `#pragma simd`.) Whole function vectorization can be achieved with the annotation `#pragma omp declare simd`, enabling the creation of multiple versions of the function that can process vectorial data from a single invocation.

Annotations can be tuned with OpenMP clauses. The `collapse` clause specifies loops to collapse with the loop immediately following the clause into one larger iteration space. The clause `safelen(N)` specifies the maximal vector length used for loop vectorization. The `simdlen(N)` clause defines the vector length of the arguments of a function annotated with `declare simd`.

4.5 Automatic vectorization

Automatic vectorization techniques detect parallelism in scalar code and transform it in order to exploit SIMD hardware [14, 15, 25–27].

A common approach is *Instruction-level parallelism* (ILP) [28–30]. This technique focuses on loop vectorization. The target loop is unrolled n times, where n is the desired vector length. In the next step, similar instructions from n consecutive iterations are replaced by SIMD instructions. Loops with unknown iteration count or early exit points make it more difficult to apply ILP, as preparation or fix-up code must be introduced. Loop-carried dependencies can prevent loop vectorization.

A different automatic vectorization approach is *Superword-level parallelism* (SLP) [31]. This technique is independent of loops, as it leverages parallelism available in a set of scalar operations:

```
e = a + b;
f = c + d;
m = e * f;
```

The first two instructions in the example are independent of each other and can be executed in parallel. SLP will merge these instructions such that only one vectorial addition is performed. This technique may also involve instruction reordering to produce vectorization candidates. A more advanced technique of SLP, called Padded SLP [32], extends SLP by inserting redundant instructions to produce additional vectorization candidates.

Chapter 5

Evaluation

5.1 Implementation

We implemented Sierra as a fork of the *LLVM*-based compiler *clang*. At the time of evaluation, our extension has been incorporated into clang version 3.3. As Sierra only adds new functionality to clang, and does not substitute any of the existing, Sierra supports the complete C++14 language standard. The Sierra extension must be explicitly enabled by supplying the command-line switch `-fsierra`. Activating the feature will not break compilation of existing C++ code.

Sierra directly operates on the AST representation of the input program. Semantic analysis propagates vector types through type conversion and detects errors caused by non-matching vector lengths. Code generation is augmented by special cases for vector types, and immediately emits vectorized LLVM code. After the source to IR translation, Sierra continues with clang's usual compilation pass.

LLVM's type legalization phase [33] may split vectors to match the target machine's native vector length. This allows to use vectors of length twice the machine's native vector length, e.g. `int varying(8)` on SSE. This technique is called *double pumping*. Our benchmarks show that double pumping may indeed result in a performance improvement.[1]

5.2 Experimental Results

We selected five benchmarks to evaluate the performance of Sierra. We had to modify the source of every benchmark to use `varying` types to trigger SIMD code generation. We exposed the vector length in the source program as a macro, such that passing `-DVECTOR_LENGTH=L` to the compiler sets the vector length of the program to `L`. As a baseline for our evaluation we used a scalar version of the program. The clue is, that the scalar version is only a special instance of the vector version; we generate it by setting the vector length to 1. Hence, any modification on the original program influences the

[1]Although we do not have evidence, we claim that increased cache-locality is the reason for this performance improvement.

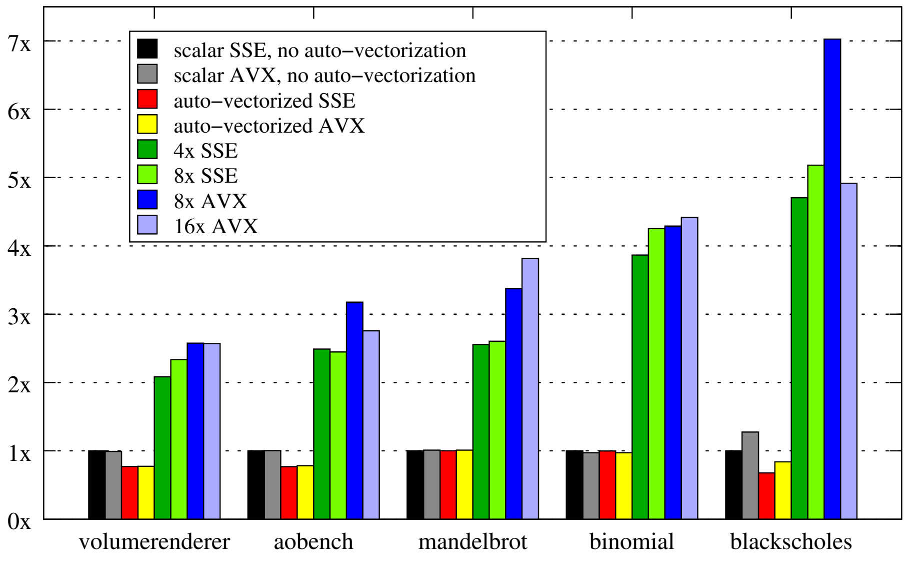


Figure 5.1: *Speedups relative to scalar, non-vectorized SSE version*

scalar and the vectorial variants equally.

We compiled several versions of all benchmarks by specifying `-DVECTOR_LENGTH`, no further modification of the source code was necessary. We compiled all programs with `-O3` and `-ffast-math`. We compiled the programs for both SSE4.2 (`-msse4.2`) and AVX (`-mavx`).

Our tests ran on an Intel® Ivy Bridge Core™ i7-3770K CPU. We used the median performance of 11 runs for computing the speedups shown in Figure 5.1.

We measured the performance of the scalar instances on both SSE and AVX. We compiled each scalar instance with and without LLVM's auto-vectorizer. The scalar SSE version without LLVM's auto-vectorizer serves as our baseline performance, and hence has a speedup of 1x. Then we instantiated vectorized versions of all programs. On SSE we used the vector lengths 4 (native) and 8 (double pumped). In case of AVX we used vector lengths 8 (native) and 16 (double pumped).

Compiling the scalar, non-vectorized version for AVX instead of SSE did not make notable differences. It is surprising, that auto-vectorization did not yield a performance improvement; in three cases, performance decreased.

5.3 Porting to Sierra

To demonstrate how to port an existing program to effectively exploit SIMD hardware using Sierra, we take a closer look at the `volumerenderer` benchmark. The two most interesting functions are displayed in Figure 5.2a and Figure 5.2c.

The function `render` contains a loop nest to iterate over all pixels of the generated image. For each pixel, `generate_ray` computes a new ray and `raymarch` is called. Finally, the value for the pixel is written to `image`.

```
void
render(float volume[], float image[], /*...*/) {
  for (int y = 0; y < image_height; ++y) {
    for (int x = 0; x < image_width; ++x) {

      auto ray = generate_ray(x, y, /*...*/);
      auto result = raymarch(volume, ray, /*...*/);
      image[y * image_width + x] = result;
    }
  }
}
```

(a) `Scalar render`

```
void
render(float volume[], float image[], /*...*/) {
  for (int y = 0; y < image_height; ++y) {
    for (int xx = 0; xx < image_width; xx += L) {
      auto x = xx + seq<L>();
      auto ray = generate_ray(x, y, /*...*/);
      auto result = raymarch(volume, ray, /*...*/);
      image[y * image_width + x] = result;
    }
  }
}
```

(b) `Vectorial render`

```
float
raymarch(float volume[], Ray& ray, /*...*/) {
  float rayT0, rayT1;
  if (!intersect(ray, bbox, rayT0, rayT1))
  return 0.f;
  // intersect initializes rayT0, rayT1

  // radiance along the ray
  float result = 0.f;

  // induction variables
  auto pos = ray.dir*rayT0 + ray.origin;
  auto t = rayT0;

  while (t < rayT1) {
    auto d = density(pos, volume, /*...*/);

    // terminate on high attenuation
    auto atten = /*...*/;
    if (atten > THRESHOLD)
    break;

    auto light = compute_lighting(/*...*/);
    result += light * /*...*/;
    pos += /*...*/;
    t += /*...*/;
  }

  return gamma_correction(result);
}
```

(c) `Scalar raymarch`

```
float varying(L)
raymarch(float volume[], Ray varying(L)& ray, /*...*/) {
  float varying(L) rayT0, rayT1;
  if (!intersect(ray, bbox, rayT0, rayT1))
  return 0.f;
  // intersect initializes rayT0, rayT1

  // radiance along the ray
  float varying(L) result = 0.f;

  // induction variables
  auto pos = ray.dir*rayT0 + ray.origin;
  auto t = rayT0;

  while (t < rayT1) {
    auto d = density(pos, volume, /*...*/);

    // terminate on high attenuation
    auto atten = /*...*/;
    if (atten > THRESHOLD)
    break;

    auto light = compute_lighting(/*...*/);
    result += light * /*...*/;
    pos += /*...*/;
    t += /*...*/;
  }

  return gamma_correction(result);
}
```

(d) `Vectorial raymarch`

Figure 5.2: *Implementation of* `raymarch` *and* `render` *in C++/Sierra. Syntactic differences between the scalar and vectorial variant are highlighted.*

The subroutine `raymarch` casts a ray through the voxel volume and accumulates the density of each hit voxel. First, it calls `intersect` to test whether the ray hits the voxel volume. If this is the case, `rayT0` and `rayT1` are set to the start respectively the end of the ray. Then a loop traverses the ray from start to end.

The highlighted text regions show the syntactic differences that are necessary to port the scalar program to Sierra. The function `seq<L>()` returns a vector of the form `{0, 1, ..., L-1}`. We can see that only eight small changes were necessary to efficiently port this program to Sierra.

Chapter 6

Future Work

6.1 goto & switch-case

Although the semantics for most vectorized C++ statements are straight-forward, the `goto` and `switch-case` statements are anything but trivial. The following statements we make about `goto` hold for `switch-case` as well.

Let us have a look at the following example:

```
loop:
...
if (cond)
    goto exit;
else
    goto loop;
exit:
...
```

This code may be nested in some arbitrarily complex context with arbitrarily many masks. The code in the example basically implements a `do`-`while` loop. The problem here is that the *then*-statement of the `if` is scheduled before the *else*-statement. The *then*-statement needs to determine at runtime whether all lines are live with respect to the current mask. Only then it is allowed to jump to `exit`. The *else*-statement however has to jump back to `loop` whenever at least one lane is live. We can see that the behaviour of a `goto` depends on whether the branch jumps backwards or forwards in the control flow graph. This is a problem that cannot be solved by syntax-guided code generation. We currently believe that at least a control dependence analysis is necessary to determine which checks must proceed a branch introduced by a `goto`.

6.2 Vectors of Pointers

Our language SLang did not include pointer types at all. Allowing pointers to vectors is straight-forward. Vectors can be stored to and loaded from memory by corresponding vector load/store instructions provided by the ISA. Even if these vector loads/stores are not provided, we can fall back to regular loads/stores. To store a vector it is disassembled into vector length many scalars, which are then stored one after another. When loading a vector we assemble it from the stored scalars.

It gets more complex as we allow vectors of pointers. When dereferencing a vector of pointers as *RValue*, each pointer has to be dereferenced separately. We need to assemble a new vector with the referenced values. Vice versa, when dereferencing such a vector as *LValue* to store values, we need to write to vector length many different locations.

As one can imagine, this imposes a lot of overhead. And even worse, this overhead is not visible to the programmer, and hence this is not in the spirit of C/C++.

There are two cases for which vectors of pointers can be implemented efficiently. The first one is uniform vectors. If every element of the vector points to the same memory region, we can issue a scalar load, and broadcast the loaded value afterwards. The other case is a vector of pointers pointing to a consecutive memory region, i.e. vectors of the form `{p, p+1, p+2, ...}`. Most ISAs provide special vector load/store instructions for these vectors. However, an analysis is required to find out when a vector is of this form.

6.3 Functions with Implicit Polymorphism in Vector-Length

Assume we want to implement a function `pow(base,exp)` to compute the power function on positive integers. Figure 6.1 shows a scalar implementation of this function.

```
int pow(int base, int exp) {
    int res = 1;
    while (exp--)
        res *= base;
    return res;
}
```

Figure 6.1: *Implementation of power*

To support this function in a vectorized context, we need to overload the function for vector arguments:

```
1 template<int L>
2 int varying(L) pow(int varying(L) base, int exp);
3
4 template<int L>
5 int varying(L) pow(int base, int varying(L) exp);
6
7 template<int L>
8 int varying(L) pow(int varying(L) base, int varying(L) exp);
```

The declaration from line 3 suffices and line 1 and line 2 are not necessary. Scalar arguments would then be broadcast on call. However, it is desirable to have special versions in case some arguments are scalar. In the example in Figure 6.1, annotating parameter `exp` with `varying` causes the loop to always be vectorized, and may impose a performance penalty if the original argument was scalar.

The number of specializations grows exponentially in the number of parameters. It is obviously bad practice to have the programmer implement all specializations of a vectorial function. Instead the programmer should only write down the scalar implementation, as in Figure 6.1, and the compiler creates all necessary vectorial versions on demand. This is a complex task and cannot be solved by naïve template meta-programming. The compiler needs to detect that the variable `res` is always vectorial, no matter which parameter is vectorial.

6.4 Vectorization of Compound Types

We already mentioned that vectorization of data-only compound types is achievable with small effort. Vectorization of complex classes however is much harder. The compiler does not only have to look at member variables, but also needs to analyze all functions that operate on these member variables. Vectorization of a member may now depend on another variable or the implementation of a function.

This task may become straight-forward when we can rely on functions with implicit polymorphism in vector-length

Chapter 7

Conclusion

This thesis presented Sierra: A SIMD Extension for C++. Chapter 2 shows how the `varying` keyword is used to declare vector types and how these types integrate with general C++. We further show that these types synergize well with both macros and templates. In Chapter 3 we present our minimalistic showcase language `SLang` and our syntax-guided code generation. We present translation patterns for vectorial `if`, `while`, and the lazy evaluation operators `||` and `&&`. With examples we show potential pitfalls in a naïve translation approach and explain how to avoid them. In Chapter 4 we compare our work with related approaches, including famous tools like OpenCL, Intel® SPMD Program Compiler, Intel® Array Building Blocks and OpenMP. We present our evaluation in Chapter 5. The results of our experimental results are shown in Section 5.2. In Section 5.3 we see the necessary differences to port one of the benchmarks to Sierra. As we expected, Sierra allows programming of readable and maintainable source code that compiles to high-performance machine code.

You can download the Sierra compiler from `www.github.com/sierra-lang/sierra`. To keep informed about the progress of this project you can visit our website at `www.sierra-lang.org`. There you will find a roadmap with future goals and all publications related to Sierra. A link to our Github site is also provided.

Bibliography

[1] *AMD64 Architecture Programmer's Manual Volume 1: Application Programming.* Publication No. 24592, Revision 3.14. Advanced Micro Devices, Inc. Sept. 2007.

[2] *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions.* Publication No. 24594, Revision 3.14. Advanced Micro Devices, Inc. Sept. 2007.

[3] *AMD64 Architecture Programmer's Manual Volume 4: 128-Bit and 256-Bit Media Instructions.* Publication No. 26568, Revision 3.10. Advanced Micro Devices, Inc. Sept. 2007.

[4] Linley Gwennap. 'Altivec vectorizes powerPC'. In: *Microprocessor Report* 12.6 (1998), pp. 1–6.

[5] *Intel®:64 and IA-32 Architectures Optimization Reference Manual.* Order Number 248966-020. Intel Corporation. Nov. 2009.

[6] *Intel®:64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture.* Order Number 253665-033US. Intel Corporation. Dec. 2009.

[7] *Intel®:64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M.* Order Number 253666-033US. Intel Corporation. Dec. 2009.

[8] *Intel®:64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z.* Order Number 253667-033US. Intel Corporation. Dec. 2009.

[9] W. Baxter and III H. R. Bauer. 'The program dependence graph and vectorization'. In: *POPL*. 1989.

[10] Pierre Estérie et al. 'Boost.SIMD: generic programming for portable SIMDization'. In: *Proceedings of the 2014 Workshop on Workshop on programming models for SIMD/Vector processing.* ACM. 2014, pp. 1–8.

[11] Roland Leißa, Sebastian Hack and Ingo Wald. 'Extending a C-like Language for Portable SIMD Programming'. In: *PPoPP*. 2012.

[12] Matt Pharr and William R. Mark. 'ispc: A SPMD Compiler for High-Performance CPU Programming'. In: *InPar*. 2012.

[13] Khronos. *The OpenCL Specification.* Version: 1.2. 2012.

[14] J. R. Allen et al. 'Conversion of Control Dependence to Data Dependence'. In: *POPL*. 1983.

[15] Ralf Karrenberg and Sebastian Hack. 'Whole Function Vectorization'. In: *CGO*. 2011.

[16] ISO/IEC. 'IEC 9899: 2011 Information technology - Programming languages - C'. In: *International Organization for Standardization, Geneva, Switzerland* (2011).

[17] Matthias Braun et al. 'Simple and efficient construction of static single assignment form'. In: *Compiler Construction*. Springer. 2013, pp. 102–122.

[18] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., 1962.

[19] Greg Michaelson and Paul Cockshott. *Vector Pascal, an array language*. 2002.

[20] Intel Cilk Plus Development Team. *Intel Cilk Plus Language Specification*. 2013.

[21] R.D. Blumofe et al. 'Cilk: An efficient multithreaded runtime system'. In: *ACM SigPlan Notices* 30.8 (1995), p. 216.

[22] Michael McCool. 'A Retargetable, Dynamic Compiler and Embedded Language'. In: *CGO*. 2011.

[23] Volker Weinberg. 'Data-parallel programming with Intel Array Building Blocks (ArBB)'. In: *arXiv preprint arXiv:1211.1581* (2012).

[24] *OpenMP Application Program Interface*. OpenMP Architecture Review Board, 2013.

[25] Randy Allen and Ken Kennedy. 'Automatic Translation of FORTRAN Programs to Vector Form'. In: *ACM Trans. Program. Lang. Syst.* (1987).

[26] Viet Ngo. 'Parallel Loop Transformation Techniques For Vector-Based Multiprocessor Systems'. PhD thesis. University of Minnesota, 1994.

[27] Dorit Nuzman and Ayal Zaks. 'Outer-Loop Vectorization: Revisited for Short SIMD Architectures'. In: *PACT*. 2008.

[28] Gerald Cheong and Monica Lam. 'An Optimizer for Multimedia Instruction Sets'. In: *SUIF*. 1997.

[29] Andreas Krall and Sylvain Lelait. 'Compilation Techniques for Multimedia Processors'. In: *International Journal of Parallel Programming* (2000).

[30] N. Sreraman and R. Govindarajan. 'A Vectorizing Compiler for Multimedia Extensions'. In: *International Journal of Parallel Programming* (2000).

[31] Samuel Larsen and Saman Amarasinghe. 'Exploiting Superword Level Parallelism with Multimedia Instruction Sets'. In: *PLDI*. 2000.

[32] Vasileios Porpodas, Alberto Magni and Timothy M Jones. 'PSLP: Padded SLP Automatic Vectorization'. In: ().

[33] Yosi Ben-Asher and Nadav Rotem. 'Hybrid type legalization for a sparse SIMD instruction set.' In: *TACO* (2013).