Extending a C-like Language for Portable SIMD Programming

Roland Leißa Sebastian Hack
Compiler Design Lab, Saarland University
{leissa, hack}@cs.uni-saarland.de

Ingo Wald
Visual Applications Research, Intel Corporation
ingo.wald@intel.com

Abstract

SIMD instructions are common in CPUs for years now. Using these instructions effectively requires not only vectorization of *code*, but also modifications to the *data layout*. However, automatic vectorization techniques are often not powerful enough and suffer from restricted scope of applicability; hence, programmers often vectorize their programs manually by using intrinsics: compiler-known functions that directly expand to machine instructions. They significantly decrease programmer productivity by enforcing a very error-prone and hard-to-read assembly-like programming style. Furthermore, intrinsics are not portable because they are tied to a specific instruction set.

In this paper, we show how a C-like language can be extended to allow for portable and efficient SIMD programming. Our extension puts the programmer in total control over where and how control-flow vectorization is triggered. We present a type system and a formal semantics of our extension and prove the soundness of the type system. Using our prototype implementation IVL that targets Intel's MIC architecture and SSE instruction set, we show that the generated code is roughly on par with handwritten intrinsic code.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; D.3.1 [*Formal Definitions and Theory*]: Semantics, Syntax

General Terms Languages, Performance, Theory

Keywords language theory, parallel programming, polymorphism, semantics, SIMD, SIMT, type system, vectorization

1. Introduction

SIMD instructions are available on commodity processors for more than a decade now. Many developers from various domains (for example, high-performance graphics [7], databases [37], or bioinformatics [6]) use SIMD instructions to speed up their applications. Many of those algorithms are not massively data-parallel but contain data-parallel and scalar parts intermixed at a fine granularity. To illustrate this, let us run through the fundamental techniques which are necessary in order to port traditional code to SIMD architectures by considering the small example in Figure 1. It is taken (and slightly modified) from a paper by Zhou and Ross about the use of SIMD instructions to speed up data base operations [37]. The program scans data records for a certain criterion and returns the value of the first record for which the criterion holds. In this particular example, we assume that at most one element in the array fulfills the criterion.[1]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
Copyright © 2012 ACM 978-1-4503-1160-1/12/02. . . $10.00

```
1  struct data_t {
2    int key;
3    int other;
4  };
5
6  int search(data_t* data, int N) {
7    for (int i = 0; i < N; i++) {
8      int x = data[i].key;
9      if (4 < x & x <= 8) return x;
10   }
11   return -1;
12 }
```

Figure 1. Example program (by Zhou and Ross [37])

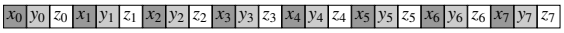
```
1  struct simd_data_t {
2    simd_int key;
3    simd_int other;
4  };
5
6  int search(simd_data_t* data, int N) {
7    for (int i = 0; i < N/L; ++i) {
8      simd_int x   = load(data[i].key);
9      simd_int cmp = simd_and(simd_lt(4, x),
10                             simd_le(x, 8));
11     int mask     = simd_to_mask(cmp);
12     if (mask != 0) {
13       simd_int result = simd_and(mask, x);
14       for (int j = 0; j < log2(L); j++)
15         result = simd_or(result,
16                          whole_reg_shr(result, 1 << j));
17       return simd_extract(result, 0);
18     }
19   }
20   return -1;
21 }
```

Figure 2. Manually vectorized example program

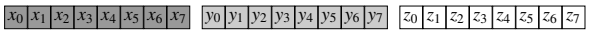
The hand-vectorized version of the program is shown in Figure 2. We assume that `L` is the native vector length of the target machine, i.e., there are `L` slots in one SIMD register and each slot can hold exactly one `int`. When performing vector computations we refer to the computations applied to a certain slot index as *SIMD* or *vector lane*. The program takes `L` data items from the array at once and processes them in parallel. It features the three most common techniques used to adapt a program for SIMD instructions:

Adapting the data layout. Fetching `L` values in parallel from memory is tremendously faster, if they lie consecutively in memory. Thus, the traditional *array of structures* layout (AoS, see Figure 3a) is not suitable for SIMD programming, as it requires non-contiguous memory access—called *scatter* and *gather* (S/G)—that severely degrades performance. A *structure of arrays* (SoA, see Figure 3b) provides a better solution. However, a loop over an SoA must maintain one pointer for each

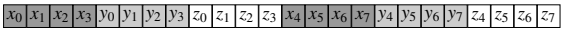
[1] Zhou and Ross mention searching in B^+ trees with unique keys as an application.



(a) Array of Structures (AoS)



(b) Structure of Arrays (SoA)



(c) Hybrid Structure of Arrays (Hybrid SoA)

Figure 3. Different array layouts with vector length 4 for a struct with three members: `x`, `y` and `z`

member. For this reason and due to data locality problems, the *hybrid SoA* (see Figure 3c) is superior, as this layout does not suffer from these problems. It emerges by first inflating the original struct to the vector length (we call this unit a *block* and the process of doing so *type vectorization*) and then grouping this block in an array. This data layout is regarded best practice in SIMD programming and is recommended by various application notes of all major CPU manufacturers, e.g. [13]. These blocks can be used to build other vectorized data structures like linked lists or trees just as well.

Control-flow to data-flow conversion. In line 9, the comparison is evaluated on all `L` loaded items in parallel. The result is a vector of booleans being *true* for those vector lanes for which the comparison was *true*. That mask has then to be converted into an ordinary `int` in order to use it in the conditional branch in line 12. In line 13, the mask is used to discard all values for which the comparison was *false*.

Seamless integration of scalar and vectorized code. While several statements of our example program perform data-parallel computations, the inner loop (lines 14–16) is *not* vectorized. It performs a scan-like operation: by successively shifting and OR-ing the *whole* SIMD variable, the first non-zero element of the vector is propagated to element 0 of the vector. It is typical for manually vectorized code to have a strong interaction between scalar and vector code.

This example illustrates the current state of the art in hand-tuning programs for SIMD instruction sets and clearly shows its limitations. First, vector code has to be written using assembly-like intrinsics: compiler-known functions that directly correspond to an assembly instruction. While operator overloading can improve the readability of expressions, control structures can usually not be overloaded. Hence, the control to data-flow conversion has to be performed manually by explicit handling of masks. This severely degrades the readability of the code because control statements are replaced by clumsy mask handling code. Furthermore, intrinsics directly expose a concrete instruction set architecture (ISA) to the programmer. Accordingly, intrinsics code is not only inherently unportable between different processor architectures but also within different revisions of the ISA *within* the same processor family.

Second, the definition of data structures is no longer compositional: To vectorize a struct, the programmer has to create a new type where the members of the struct are vectorized (see Figure 2). Often she is interested in the scalar *and* the vectorized type because she often refers to a single component of a vector. Hence, all types would have to be written twice. However, it is reasonable for the programmer to assume that she can obtain the vectorized version of a data type `T` by applying a type constructor, say `T block[N]`, which yields the desired block type. Furthermore, the compiler *knows* that `T block[N]` is the `N` times vectorized version of `T`. This is important for parametric polymorphism (see below) and correct handling of S/G in the type system.

Third, the hand-vectorized function of Figure 2 works *only* in a vector context. Many functions however should work in a scalar *and* in a vector context. Hence, functions should be polymorphic with respect to vectorization where possible.

We believe it to be clear that the code transformations presented above cannot be carried out automatically. Current optimizing compilers vectorize loop nests which have to satisfy strong prerequisites. Automatic interprocedural data-structure rewriting or automatic scalar/vectorial code switching are out of the scope of what an optimizing compiler for a traditional C-like language can do. Furthermore, porting an algorithm to SIMD instructions does not only change the mapping of a *given* implementation to another hardware but often requires adapting the implementation of the algorithm itself. For example, note lines 13–17 in Figure 2 differ from the corresponding code in Figure 1.

On GPUs, which internally also make extensive use of SIMD, this issue is addressed by not using a traditional scalar programming paradigm like in C, but rather using the SIMT paradigm (single instruction, multiple threads) as employed in CUDA [30] or OpenCL [17]: In the SIMT paradigm each program—called *kernel*—is always instantiated n times to n logical *threads*. These threads all execute this same program at the same time but on different data; SIMD is then employed by always processing k such threads together where k is the SIMD width. A group of k threads is also called *warp*. Each of a warp's k threads run one SIMD lane. In this programming paradigm, the user writes scalar-looking, C-like code, and the control-flow to data-flow conversion along with the necessary predication is mostly handled by the hardware.

While SIMT is a very powerful paradigm that allows to express parallelism in a scalar-looking way, it also has its disadvantages. In particular, in a SIMT language *everything* is always vectorized, which closely matches the execution model of such GPUs, but does not easily allow to express scalar parts of a program. Executing truly scalar code then requires explicitly masking off all but one thread of a block and costly explicit synchronization between the warps in a block, since the hardware usually needs to run multiple such warps in parallel. Furthermore, by explicitly *hiding* the effects of vectorization, these languages have no means of allowing the programmer to define vectorized data layouts, thus leaving her with the choice between inefficient AoS data layouts (and costly AoS to (hybrid) SoA conversions on access), or manual vectorization of data structures. Above all, writing all the glue code in order to merge the host code with the kernels is very inconvenient.

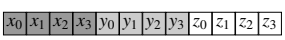
1.1 Contributions

In this paper, we propose a different programming model for data-parallel programming that combines the advantages of both scalar and SIMT programming, and is especially suited for machines with explicit SIMD instructions. We do this by defining a small core language called VecImp (see Sections 2 and 4), which extends the type system and formal semantics of an existing formalization of a substantial subset of C, called Imp (see Section 3).

- In contrast to the SIMT model, we do not require the strict separation in host code and kernels but integrate vectorization seamlessly into the language. Moreover, we allow to mix scalar and SIMT programming (and scalar and vectorized data structures) in the same basic blocks.
- We provide a type qualifier `block[N]` to obtain a block of vector length `N`. The example code

```
struct vec3 { float x, y, z; };
vec3 block[4] v;
```

  produces the following memory layout for `v`:



- In contrast to the vectorize-everything paradigm of SIMT, a VecImp program starts in a *scalar* context. We use the type

system of VecImp to trigger vectorization: An if-statement with a vector-type condition is vectorized.

```
int block[4] v, w;
if (v < w) /* vectorized */ else /* vectorized */
```

- We implemented several features of VecImp in our research language IVL and evaluated the performance of the vectorized code experimentally on several benchmarks. Our target machine was a 16-wide Intel® Many Integrated Core (MIC) processor [14, 32]. We are within 4% to 15% of the performance of hand-written intrinsic code. Writing the IVL code was almost as easy as writing traditional scalar code. Besides that, IVL code is fully portable because it does not contain any architecture-dependent details (see Section 5). The programs can just be recompiled for the SSE back-end.

2. VecImp at a Glance

As outlined in the introduction, a programmer often wants to perform SIMD operations only on certain lanes and not on the whole vector. We call these lanes *active lanes*. In hardware, this is either accomplished by *predicated execution* [32] or *blending* (SSE, AVX). In the first case, each vector instruction is equipped with a mask, which indicates active lanes, and a vector instruction performs the operation only for active lanes. In the latter case, vector instructions always perform operations for all lanes. Unwanted effects must be later on masked out by so-called blend operations. The programming model of VecImp exposes predicated execution to the programmer, although blending can be used by the compiler in order to emulate this execution model. For this reason, every statement in VecImp is executed in a *scalar* or *vector context*. For typing, the context corresponds the vector length of the current predication mask—also called *current mask*. The programmer has read access to this variable by the `current_mask` keyword. A context of length one is called *scalar context*. In VecImp, evaluation begins in a scalar context and control-flow statements like `if` or `while` spawn different vector contexts:

```
// vector length of context = 1; current_mask = T
int  block[4] v = <0,3,4,1>;
int  block[4] w = 3;      // <3,3,3,3> via broadcast
bool block[4] m = v < w; // <T,F,F,T>
++v;                     // <1,4,5,2>
if (m) {
  // vector length of context = 4; current_mask =  m
  v += 2; // <3,4,5,4>
} else {
  // vector length of context = 4; current_mask = ~m
  v += 3; // <3,7,8,4>
}
// vector length of context = 1; current_mask = T
```

As can be seen in this example, a type modifier `block[N]` is available. This qualifier recursively inflates the data type to be of vector length `N`. If a qualifier is elided, the variable is *unbound*. This means that the vector length is inferred by the vector length of the right-hand side. If a right-hand side does not exist, the vector length of the current context is used:

```
struct vec3 { float x, y, z; };
// scalar context
vec3 block[4] v;
vec3          w = v; // vec3 block[4];
vec3          x;     // vec3 block[1];
```

Let us now implement a dot product:

```
float dot(vec3 a, vec3 b) {
    return a.x*b.x + a.y*b.y + a.z*b.z;
}
```

All *unbound* parameters of functions in VecImp are polymorphic in their vector length and so are `a` and `b`. The vector length of an *unbound* return type is automatically inferred. Thus, a call-site in a scalar context

```
1  struct data_t {
2    int key;
3    int other;
4  };
5
6  int search(data_t *scalar data, int scalar N) {
7    int L = lengthof(*data);
8    for (int i = 0; i < N/L; ++i) {
9      int x = data[i].key;
10     if (4 < x & x <= 8)
11       int block[L] result = [x, 0];
12       scalar {
13         for (int j = 0; j < log2(L); ++j)
14           result |= whole_reg_shr(result, 1 << j);
15         return get(x, 0);
16       }
17   }
18   return -1;
19 }
```

Figure 4. Example program in VecImp

```
vec3 block[4]  a = /*...*/;
vec3 scalar    b = /*...*/;
float block[4] r = dot(a, b);
```

instantiates a version of `dot` with this signature:

```
float block[4] dot(vec3 block[4] a, vec3 scalar b);
```

If a scalar and a vectorial value are mixed in an operation, the scalar value is broadcast. Thus, the uses of `b.x`, `b.y` and `b.z` get implicitly broadcast in this version. But what happens, if we call the same function from a vectorial context?

```
if (a.x < b.x) { // enter context of length 4
    float block[4] r = dot(a, b); /*...*/
}
```

This time `dot` must know of the current mask. For this reason, we pass the current mask as hidden parameter to `dot`. In VecImp, functions are also polymorphic in the context they are called in. Thus, the signature of this `dot` instance is:

```
float block[4] dot(bool block[4] current_mask,
                   vec3 block[4] a, vec3 scalar b);
```

As a more sophisticated example, let us reconsider the introductory example in VecImp (Figure 4). The function `search` can be invoked with a pointer to an ordinary array of `data_t` or a hybrid SoA of `data_t`. However, the pointer itself is scalar, denoted by the `scalar` modifier which is just shorthand for `block[1]`. Hence, one cannot pass a vector of pointers to `search`. Similarly, `N` stays scalar in all possible instances of `search`. Consider the call-site in a scalar context:

```
data_t block[4] *data; /*...*/
int x = search(data, N);
```

The type of `data` is an unbound pointer to a hybrid SoA of length 4 of `data_t`. Since `data` is declared in a scalar context, `data` itself is instantiated as scalar while the referenced type remains vectorial. Since `search` is invoked from a scalar context, the context of this particular `search` instance created for this call-site is scalar, too. As `L` and `i` in line 7 and 8, respectively, inherit the vector length from the right-hand side, these variables are scalar. Now consider line 9: Since the type of the right-hand side is `int block[4]` and `x` is unbound, `x` is also of vector type `int block[4]`. As `x` is now of vector type, the if statement in line 10 is vectorized. Thus, reading from `x` just obtains the (by precondition unique) found key which is—via the assignment in line 11—written into the vector `result`. However, the inactive lanes of `result` would stay undefined and we therefore set them explicitly to `0`. This is important because subsequently, the vector `result` is processed as a whole and not in a vectorized mode: The algorithm propagates the found key to position 0. For this reason, we want to enter a scalar context again. The `scalar`

keyword in front of a statement stores the current mask on a stack and resets it to the scalar mask. Hence, the for loop beginning in line 13 is executed in a scalar context. After the for loop finishes execution, the scalar element at position 0 of the vector is returned.

In the case that `search` is invoked on scalar data, the whole procedure degenerates to a scalar program whose semantics is equivalent to the program in Figure 1. Although we cannot gain an optimized vectorial program from a scalar one, we can still extract the scalar version from a program written in a polymorphic way.

3. Formalization

In this section, we first introduce the notation used throughout this paper and then the imperative language Imp.

The set which holds all powers of two is referred to as $\mathbb{P} \coloneqq \{2^i \mid i \in \mathbb{N}\}$. We use $\overline{a}$ as shorthand for a sequence $a_0, \ldots, a_{|\overline{a}|-1}$ with zero or more elements, while $|\overline{a}|$ denotes the number of items in a sequence. The term $\overline{a}^+$ represents a sequence with one or more elements. Sequences are propagated into more complex expressions while global values and constants remain untouched. Thus, $(\overline{a, 3})$ expands to $(a_0, 3, \ldots, a_{|\overline{a}|-1}, 3)$.

3.1 Types

We use calligraphic letters to indicate sets of types and use small Greek letters for types. The size of a type τ given in bytes is referred to as $|\tau|$. Furthermore, we make use of the following conventions:

We use standard C-like primitive integer and floating point types whose sizes (in bytes) must be a power of two. Booleans are denoted by $\texttt{bool} \coloneqq \{\mathsf{T}, \mathsf{F}\}$. The size of a `bool` is implementation-dependent and not transparent to the programmer. We pool these types into a set of *atomic types* $\mathcal{A}$.

A pointer τ* points to a location which is interpreted to be of type τ; the size of a pointer is also implementation-dependent.

A record $\rho \coloneqq \{\overline{\tau}^+\}$ is compounded of other element types $\overline{\tau}^+$. A record may not (in)directly contain an element of its own type, but cycles induced by pointers are allowed (as in C). A projection $\rho.i$, where $i \in \mathbb{N} \wedge 0 \leq i < |\overline{\tau}^+|$, grabs τ_i. An implementation is free to add additional padding space between the elements of a record. This implies that the size of a record is machine dependent, too. The set of all types which can be built with atomic types, records and pointers is denoted by $\mathcal{T}$.

A *vector type* ν with element type $\tau \in \mathcal{A} \cup \{\pi* \mid \pi \in \mathcal{V}\}$ and $l \in \mathbb{P}$ elements is written as $\langle \tau \mid l \rangle$. An instance of such a type ν is written as $\langle \overline{e} \rangle$. The set of all types which can be built with scalar types and vector types is denoted by $\mathcal{V}$.

3.2 Imp

Before describing the vector extensions, we briefly review the imperative language Imp we extend. Imp is a substantial subset of C. Its formalization is closely related to Norrish [27].

The syntax is given in Figure 5. To simplify matters, we assume that each function has exactly one return statement and function calls are used instead of typical in- and prefix operators (although we silently make use of operators in later VecImp examples, for the sake of readability; moreover, we use named structs instead of tuples and other syntactic sugar for the same reason). A representative set of type inference and evaluation rules is given in Figure 7. The complete set of rules is contained in the full version of the paper [20, appx. A].

Typing rules. Static semantics maintains a type map Γ, which maps identifiers to types, and has read-access to a function map Φ, which maps identifiers to a function's signature and body. The type map is altered in a declaration (see T-Decl). On the other hand the occurrence of a variable is resolved by a lookup of the given identifier in this map (see T-Var). Type inference rules must differentiate between statements and expressions. Statements yield the dummy type ϵ. Since expressions can return lvalues, type inference rules acknowledge this fact by an internal lvalue type ${}^l\tau$ (read *lvalue of type* τ). If we want to state that a type must not be an lvalue, we write ${}_r\tau$. In many typing rules both lvalue and value types are allowed in a given premise. We annotate such a type as ${}^l_r\tau$ as shorthand (see T-Assign).

e	::=		*Expressions:*
		$\texttt{c}_\alpha$	constant of type $\alpha \in \mathcal{A}$
	|	`id`	variable
	|	e`.i`	projection
	|	`&`e	address
	|	`*`e	dereferencing
	|	e_1`[`e_2`]`	indexing
	|	`id(`$\overline{e}$`)`	invoke
s	::=		*Statements:*
		`;`	skip
	|	`{` $\hat{s}$ `}`	scope
	|	e`;`	expression-statement
	|	t `id;`	declaration
	|	e_2 `=` e_1`;`	assignment
	|	t `id =` e`;`	initialization
	|	`if (`e`)` s_1 `else` s_2	if-else
	|	`while (`e`)` s	while
$\hat{s}$	::=		*Scoped statements:*
		s	statement
	|	$\hat{s}$ s	sequence
t	::=		*Type:*
		$\alpha \in \mathcal{A}$	base
	|	t*	pointer
	|	$\{\overline{t}^+\}$	record
f	::=		*Function:*
		ϵ	empty
	|	f_1 f_2	sequence
	|	t `id(`$\overline{t\ \texttt{id}}$`) {` s `return` e`; }`	function

Figure 5. Syntax of Imp

Evaluation rules. Dynamic semantics uses small step semantics evaluating a program a in scope σ and state M in one step to a program a' in scope σ' in a new state M' with the help of the function map Φ. The state M is a memory map, mapping an address to an `i8`. We assume M to be infinite and initialized with random values. A scope σ consists of an address map A which maps an identifier of a variable to the starting address in memory and a type map T which maps an identifier to a type. In order to extract a subsequence of n bytes starting at address/offset a of a memory map M or a byte sequence m, we use $M\langle a, n\rangle$ or $m\langle a, n\rangle$ as shorthand, respectively.

As already outlined, expressions can evaluate to values and lvalues. A value consists of a sequence of bytes m and a type τ and is written as $\mathcal{V}(m, \tau)$. An lvalue $\mathcal{L}(a, \tau)$ consists of an address a and a type τ and represents a reference to a memory location. Both constructs are added to the syntax (but are not directly usable by the programmer and hence do not appear in Figure 5). For instance, E-Var looks up the identifier of a variable in the address and type map to build an lvalue. Statements can evaluate to the final configuration "`;`" (read *skip*).

A declaration (see E-Decl) allocates memory and updates the type and address maps accordingly. The function $\text{alloc}(\sigma, \tau)$ is used for this task. It returns an address pointing to $|\tau|$ bytes of usable memory. An assignment (see E-Assign) updates the memory map at the position pointed to by the given left-hand side lvalue with the given right-hand side value.

In order to memorize an old scope σ when entering a new one we use the special construct "$\{\ s\ \}^\sigma$" which is also added to the "internal" syntax (see E-InScope and E-OutScope).

e	::=	...	*Expressions:*
	|	`current_mask`	current mask
s	::=	...	*Statements:*
	|	`scalar` s	scalar
t	::=		*Type:*
		$\alpha \in \mathcal{A}\ q$	base
	|	t`*` q	pointer
	|	$\{\overline{t}^{+}\}\ q$	record
q	::=		*Qualifiers:*
		ϵ	unbound
	|	`block`	auto block
	|	`block[i]`	block

Figure 6. Syntax changes of Imp in order to obtain VecImp: The current-mask expression and the scalar statement are added. Types follow a different grammar along with the new qualifier rule.

Theorem 1 (Soundness of Imp)**.** *Each well-typed* Imp *program is either in its final configuration or*

1. *there exists an evaluation rule which progresses evaluation and*
2. *the resulting program after application of an evaluation rule preserves well-typedness.*

See the full version of the paper for details and the complete proof of this theorem [20, appx. A].

4. Extending Imp

In the following sections, we show how to extend Imp to its vector counterpart VecImp. The needed modifications of Imp's syntax in order to obtain VecImp are given in Figure 6. During discussion it is important to understand that VecImp is just a substantial intentionally small calculus. Practical languages may add as much syntactic sugar as needed. Furthermore, languages with powerful abstraction mechanisms can build higher-level vector programming constructs (see Section 7).

An excerpt of the typing and evaluation rules, which are referred to in this section, is shown in Figure 8. We have already seen in the introductory example that we must keep track of the current mask while doing SIMD computations. This has several implications. First, all typing rules need a new variable $\Lambda \in \mathbb{P}$ called *current vector length* or *context*. Likewise, all evaluation rules need a new scope variable $P_\sigma \in \langle \texttt{bool} \mid l \rangle$ ($l \in \mathbb{P}$), which reflects predication. We extend our notation of extracting a byte sequence to a predicated extraction $M\langle P, a, n\rangle$ or $m\langle P, a, n\rangle$, respectively, where P is a mask. This *current mask* or *predication variable* can be queried via the `current_mask` keyword, which is added as expression to the syntax. Generally, a statement is only executed if the current mask is not entirely false. In the special case that $\Lambda = 1$ and accordingly $P_\sigma = \langle \mathsf{T} \rangle$, we are in a scalar context.

4.1 Types

The context influences the generation of types while the programmer retains tight control over this process with the help of few qualifiers (see T-, E-DeclL). Type vectorization is a recursive process:

- A `block` type adopts the vector length of the current context, i.e., in a scalar context the type stays scalar.
- A `block[i]` type always receives vector length `i` where `i` is a compile time constant. The special case `i` $= 1$ means a scalar type and we write `scalar` as syntactic sugar for `block[1]`. In the case that `i` > 1 does not match the current vector length, a type error occurs.
- Types without explicit qualifier are qualified as *unbound*. These types adopt the vector length of the current recursion. Top level unbound types either infer the vector length from the right-hand side (in an initialization, see T- and E-Init) or adopt the current vector length (in a declaration statement, see T- and E-Decl).
- Records are never vectorized. Instead the qualifier of a record controls recursively vectorization for *unbound* element types.

The type constructor of a qualified type t in context Λ is defined as follows:[2]

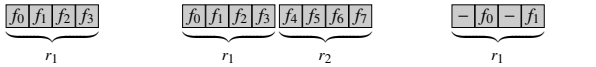
$$\mathrm{vec}_\Lambda : t \times \mathbb{P} \to \mathcal{V} \cup \{\bot\}$$

$$(\upsilon, n) \mapsto \begin{cases} \bot & \text{if } \upsilon = \tau \in \mathcal{T}\ \texttt{block[}i\texttt{]} \ \wedge\ \Lambda \neq i > 1, \\ \langle \alpha \mid n \rangle & \text{if } \upsilon = \alpha \in \mathcal{A}, \\ \langle \alpha \mid \Lambda \rangle & \text{if } \upsilon = \alpha \in \mathcal{A}\ \texttt{block}, \\ \langle \alpha \mid i \rangle & \text{if } \upsilon = \alpha \in \mathcal{A}\ \texttt{block[}i\texttt{]}, \\ \langle \mathrm{vec}_\Lambda(\tau, n)\texttt{*} \mid n \rangle & \text{if } \upsilon = \tau\texttt{*}, \\ \langle \mathrm{vec}_\Lambda(\tau, n)\texttt{*} \mid \Lambda \rangle & \text{if } \upsilon = \tau\texttt{* block}, \\ \langle \mathrm{vec}_\Lambda(\tau, n)\texttt{*} \mid i \rangle & \text{if } \upsilon = \tau\texttt{* block[}i\texttt{]}, \\ \{\overline{\mathrm{vec}_\Lambda(\tau, n)}^{+}\} & \text{if } \upsilon = \{\overline{\tau}^{+}\}, \\ \{\overline{\mathrm{vec}_\Lambda(\tau, \Lambda)}^{+}\} & \text{if } \upsilon = \{\overline{\tau}^{+}\}\ \texttt{block}, \\ \{\overline{\mathrm{vec}_\Lambda(\tau, i)}^{+}\} & \text{if } \upsilon = \{\overline{\tau}^{+}\}\ \texttt{block[}i\texttt{]}. \end{cases}$$

Consider the following example:

```
struct Bar { int c; int block d; };
struct Foo { int a; int block b; Bar scalar bar; };
// current vector length = 4
Foo foo;
int i = 0;
```

Upon the declaration of `foo` $\mathrm{vec}_4(\mathrm{Foo}, 4)$ is applied in order to vectorize the type. Member `a` gets vectorized as it sees 4 as parameter; `b` gets vectorized, because it sees 4 as current vector length; however, `c` stays scalar, because it sees length 1 passed to vec_4 via `Bar scalar bar`; finally, `d` gets vectorized for the same reason as `b`. The variable `i` stays scalar as the right-hand side is also scalar.



(a) $4 \times$ `float` (b) $8 \times$ `float` (c) $2 \times$ `float`

Figure 9. Different packagings of `floats` into hardware SIMD registers of 16 byte width

In order to exploit SIMD hardware efficiently, it is very important to organize data in a way that maximum coherence is achieved, i.e., optimally all SIMD lanes are occupied and none lie idle. If the desired vector length is equal to the one which is available in hardware, all elements can be put seamlessly into one register (Figure 9a). If the desired vector length exceeds the native one, several registers must be allocated (Figure 9b). In the case that the requested length is smaller than the native one, padding space must be inserted. One possible solution is depicted in Figure 9c, albeit other possibilities may be more advantageous for the underlying hardware. As case b) usually increases register pressure and case c) decreases throughput, it is important to choose the vector length wisely. A compiler can provide a built-in mechanism for querying the preferred vector length of a type for the given target machine:

```
struct T { double d; float f; }
T block[preferred_lengthof(T)] t;
```

See Section 7 for additional suggestions in order to abstract away other low-level details.

The current vector length also restricts types that may occur. *Access* of type τ in context Λ is only granted in a scalar context, *or* if the type is scalar, *or* if the current vector length and the one of

[2] Note that cyclic dependencies in types induced by pointers can be resolved by first defining an opaque type and not to follow that pointer and later on refining that field.

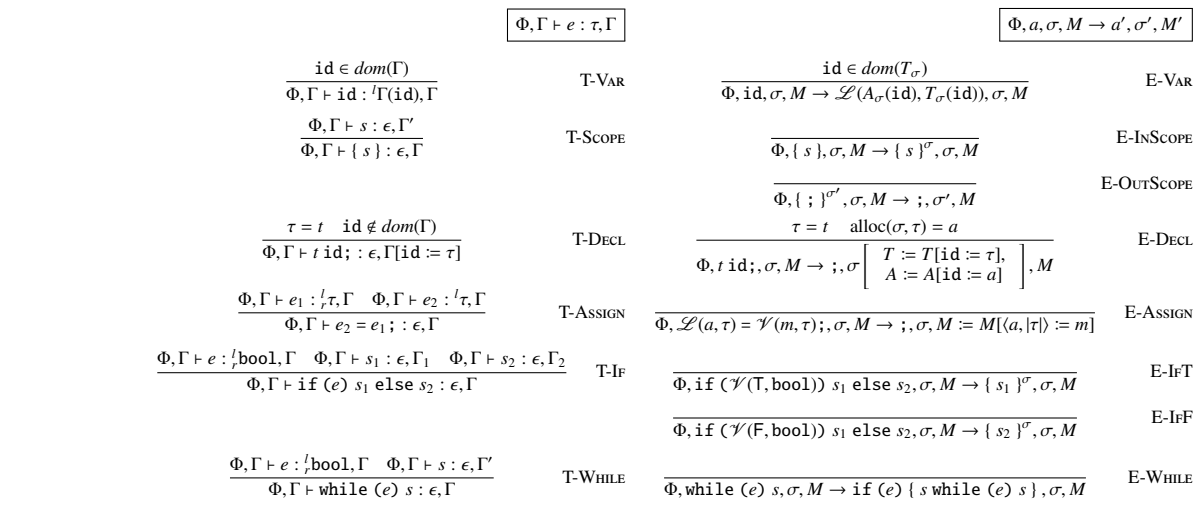


Figure 7. Static (left) and dynamic (right) semantics of Imp (excerpt)

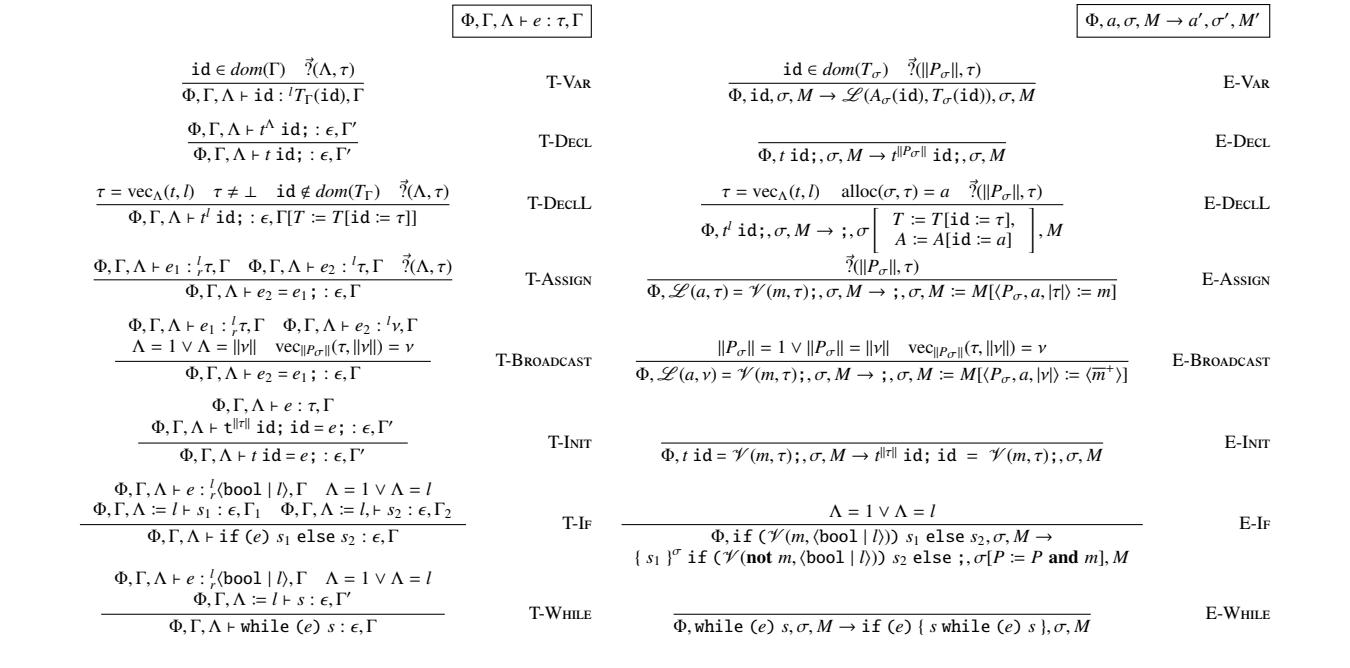


Figure 8. Static (left) and dynamic (right) semantics of VecImp (excerpt)

the type match:

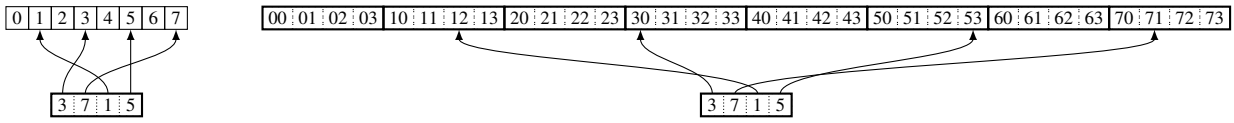
$$\vec{?} \colon \mathbb{P} \times \mathcal{V} \to \texttt{bool}$$
$$(\Lambda, \tau) \mapsto \Lambda = 1 \ \vee \ \|\tau\| = 1 \ \vee \ \Lambda = \|\tau\|$$

where $\|\tau\|$ denotes the vector length of type τ. Moreover, a scalar variable on the right-hand side of an assignment can be *broadcast* to a vector variable on the left-hand side. This is accomplished by replicating the given value to all lanes (see T- and E-Broadcast).

4.2 Triggering Control-Flow Vectorization

Conditionally executing code is usually achieved by testing a boolean and either running one or another code path. Imp supports typical if- and while-statements. In VecImp we enhance these constructs by allowing boolean vectors as conditional expressions as well. Generally, the vector length of the header must either match the one of the current mask, or a scalar conditional occurs in a vectorial context, or the conditional is in a scalar context. The latter case triggers vectorization. This means that control-flow is converted to data-flow which gives the programmer the illusion that different SIMD lanes are executed on different control-flow branches. Care must be taken when different branches have side-effects like I/O-accesses. These side-effects may happen in an unexpected order. For this reason, it is important that the programmer at least roughly understands, how the control-flow vectorization is accomplished. For the sake of simplicity, we concentrate on simple structural control-flow in this paper. For vectorization of arbitrary control-flow we refer to Karrenberg and Hack [16].

If/Else. In VecImp, both the then-block and the else-block are evaluated (see T-If and E-If). In the then-block the current mask and the test mask are combined with an AND operation whereas



(a) `int scalar* block[4] p;` (b) `int block[4]* block[4] p;`

Figure 10. Involved address calculation when indexing a vectorial pointer `p` with an index vector `int block[4] x = <3,7,1,5>`. A read `p[x]` obtains in case a) `<3,7,1,5>` and in case b) `<30,71,12,53>`. Note that both variants yield an `int block[4]`.

the else-block uses the complementary test mask. Note that an if statement degenerates to standard Imp semantics if an ordinary boolean is used in the header, since statements are not executed if the predication variable is completely false.

While. We define the semantics of the while-construct as a recursive program transformation to an if-statement containing a while-statement (see T-While and E-While). If every element of this mask evaluates to false, the body is not executed at all. Otherwise, the body must be entered while adjusting the current mask. Analogously to the if-statement, a while-statement degenerates to an ordinary while-statement when testing a scalar boolean.

4.2.1 The Scalar Statement

Sometimes it is desirable to deactivate vectorization within a kernel and proceed with scalar computations (see the introductory example in Section 1). This can be realized by adding a *scalar statement* to the syntax. This statement saves the current mask, sets the current mask to the scalar *true* value, executes the inner statement in a scalar context, and restores the saved mask afterwards. Using the scalar statement, several sequential patterns of algorithms using SIMD instructions can be implemented.

Reactivating the old mask. Although the old mask is reset after termination of the scalar statement, it may be nevertheless useful to temporarily reactivate the mask within this statement. This can be achieved by manually saving the current mask before executing the scalar statement and using the saved mask in an if-statement. Since a store just sets the active lanes, it is also important to set the inactive lanes manually to *false*. The example in Figure 4 makes use of some syntactic sugar in order to initialize elements in both active and inactive lanes. This syntactic sugar

```
T var = [true_values, false_values};
```

translates to:

```
bool mask;     // a mask; we are in a vector context
scalar { mask = false; } // set all lanes to false
mask = current_mask;     // set all active lanes to true
T var;
scalar {
   if (mask) var = true_values;  // set active lanes
   else      var = false_values; // set inactive lanes
}
```

Scalarize each active lane. Using a consecutive bit scan over the old mask can yield the index of each prior active lane. In the following example each active component of `v` of type `T block[N]` is extracted:

```
bool mask = [current_mask, current_mask];
scalar
  for(int i=bitscan(mask,0); i>=0; i=bitscan(mask,i+1)) {
    T s = get(v, i);
    //...
  }
```

A `bitscan` scans a given mask, beginning at the given index, from left to right for the next entry set to *true*. In order to indicate the case that no further *true* follows, a negative value is returned.

The loop body can now do scalar computations with `v`, broadcast this value in order to vectorize another computation, or collect such values in another data structure which can be later on used for further vectorizations, for instance.

Uniquely scalarize each active lane. A more sophisticated technique only extracts unique values from active lanes. In the example above the following adjustment of the mask in each iteration before the next `bitscan` blends out duplicates:

```
mask &= s != v;
```

4.3 Address Calculation

Since both a pointer itself and its referenced type may be scalar or vectorial, four possibilities emerge: Scalar pointers to scalar data are ordinary pointers as known from C. Scalar pointers to vectorial data integrate seamlessly into Imp as the layout of the referenced type is irrelevant for typing and semantics. However, when dealing with vectorial pointers, the programmer must use an index vector. Vectorial indexing to scalar data effectively performs a S/G (see Figure 10a) and obtains a vector. Vectorial indexing to vectorial data is more involved as depicted in Figure 10b: Each lane i "sees" the i^{th} component of a vector located at address `get(p, i)` + `get(x, i)` and hence read/write access with this vectorial indexing gathers/scatters an `int block[4]`.

4.4 Parametric Polymorphism

An important feature of VecImp is that all functions are parametric towards vectorization. This means that all unbound types which appear in the signature and the vector context itself are parametric. Upon invocation a new version of the function is instantiated (like templates in C++). This has several consequences:

- Type checking and evaluation is done with the current context/mask. This changes the vector length of all `block` variables and unbound variables in declaration statements.
- All unbound function arguments try to adopt the vector length of the parameter (like an initialization).
- If the return type is unbound the type checker infers its vector length from the return expression.
- As the semantics of all syntactical constructs depends on the instantiated types, the behaviour of the function is also dependent on these parameters.

4.5 Soundness

All these additions and changes to Imp do not harm the type system and using the same definition of soundness as in Theorem 1 (see the full version of the paper for details [20, appx. B]) we can state:

Theorem 2 (Soundness of VecImp)**.** *VecImp's type system is* sound.

5. The IVL Vectorizing Language

IVL is a prototypical R&D compiler that applies many of the proposed concepts to generate vectorized code for the Intel® MIC architecture and SSE instruction set.

5.1 IVL's Back-Ends

MIC is a many-core x86 architecture in which each x86 core is augmented with a 16-wide vector unit accessed through a rich vector instruction set. MIC supports both S/G as well as efficient masking/predication via a separate set of 16-bit mask registers.

In a vectorized context IVL emits MIC code that runs on `L` lanes in parallel, while all *scalar* constructs are executed in the core's scalar pipe, and scalar data is held in scalar registers. Rather than allowing arbitrary vector lengths, IVL currently only supports a small set of vectorization lengths per compilation unit. In particular, on MIC we currently support 16-wide and ("double-pumped") 32-wide vectorization for MIC (with 16 being the default). The SSE back-end supports 4 wide, and a future Intel® AVX back-end will also support 8 wide vectorization.

Note that MIC and SSE instruction sets fundamentally differ in many aspects: SSE and MIC have different native vector lengths, SSE does not support hardware accelerated S/G and uses blending in order to emulate predicated execution. Given our experience in implementing the SSE and MIC back-ends, we think implementing additional back-ends supporting other SIMD CPU types and ISAs is straightforward.

IVL acts as source-to-source compiler in the sense that it translates IVL code to a special "C++ with intrinsics" code that is then passed to the official Intel® C/C++ compiler [11] (`icc`) for MIC. This allows the programmer to visually inspect (and possibly modify) the emitted code, and to use this code with tools like debuggers, performance analyzers, etc.

5.2 Supported Types and Language Constructs

IVL currently supports a significant subset of C and a small set of additional keywords to guide vectorization. In terms of types, IVL fully supports bools as well as (32-bit) ints, uints, and floats, but currently only partially supports 8-bit, 16-bit, and 64-bit data types. IVL also fully supports structs, arrays, and references (including vector references to vectorized types), but only partially supports pointers.

As IVL currently only supports one global vector length `N` per compilation unit, VecImp's `block[N]` is called **`varying`**, and `block[1]` is called **`uniform`**. Like VecImp, IVL supports *unbound* variables, and full polymorphism in parameter type length. In terms of control-flow IVL supports all of **`if`**/**`else`**, **`do`**, **`while`**, **`for`**, **`break`**, **`continue`**, and **`return`**, with the sole exception of **`goto`**. In addition, IVL also supports some simple reduction operations like "**`varying bool`** `any`(**`varying bool`**)" or "**`varying bool`** `none`(**`varying bool`**)" in order to determine whether at least one or no element in a boolean vector is *true*.

5.3 On-demand Vectorization

Vectorization in IVL is done *on demand*: Similar to the behavior of templates in C++, IVL parses struct and function definitions but does not emit anything until *instances* of those types and functions are required. For example, when IVL encounters a global variable of a vectorized type, it emits a vectorized form of this type (in C++ code). Vectorization of code is triggered when IVL encounters a function with the **`kernel`** keyword; it will then emit a C function for this kernel (plus some additional helper functions to allow calling this function from the host machine if required), and will vectorize this kernel's body, which in turn will on-demand emit all functions called by this body, etc.

Statements and expressions operating on varying types will emit vector intrinsics, while purely uniform expressions/statements *will* emit only scalar code even when inside a vectorized function (like proposed by VecImp). This is highly desirable in that it uses precious vector registers and costly vector instructions only where required, and enables a mix of scalar and vector expressions that MICs superscalar architecture (with parallel scalar U and vector V pipes) is particularly good at. For example, if a for-loop in a (vectorized) function uses a loop condition that only depends on a uniform function parameter that respective loop control code will use only scalar x86 instructions.

Polymorphism in vector length. Polymorphism also works like proposed in VecImp: Unbound function arguments are bound when IVL encounters its call site, and are then bound to the vector length of the respective argument. This means that the same *logical* function can end up being emitted in multiple incarnations (depending on which of its parameters are uniform or varying). This, too, is highly desirable in that information about which data are uniform is *preserved* across function calls—usually leading to significantly more efficient code and storage than if all scalar data had been broadcast to vector form upon the first function call (as would happen in a pure SIMT paradigm). In particular, a given control-flow statement in an input function can get emitted in scalar form in one function instance, and vectorized in another.

Like polymorphism in C++, this mechanism requires the function to be known at compile-time; to use a function across different compilation units one either has to explicitly specify all parameters' vector lengths, or explicitly instantiate this function in one compilation unit.

5.4 IVL Examples

Though a full performance analysis is beyond the scope of this paper, we give a brief overview over some examples realized with our current IVL compiler, all running on a 32-core 900MHz Knights Ferry prototype board. Since IVL also has an SSE back-end, the exactly same examples will also run on any SSE-enabled architecture, but for the sake of brevity we restrict ourselves to only the Knights-Ferry results.

Proof-of-concept examples. We ported `Mandelbrot`, `nbody`, and `VolumeRender` from the CUDA SDK. These examples ran more or less "out of the box". Since IVL does not currently support any native hardware-texturing, the `VolumeRender` example has to resort to "manual" tri-linear interpolation to sample the volume, but nevertheless already reaches roughly 30 frames per second. For all other examples, both ease of porting and resulting performance matched or exceeded expectations. As just one example, the publicly available `aobench` benchmark [3] required only trivial modification to port to IVL,[3] while rendering a 1024^2 frame (with 16 samples per pixel and 16 rays per sample) in an impressive 402ms.

Ray tracer examples. As a somewhat more challenging example, we also implemented an IVL ray tracer with various shaders *into* an existing Knight's Ferry ray tracing system [36]. The IVL-based traversal, intersection, and shading code was linked together with manual intrinsics code for data structure construction and other renderers. In this setting, IVL and manual C code were actually sharing the same data structures! First, we integrated an intentionally simple *eyelight* shader into the framework. As a next step, we included an *ambient occlusion* (AO) renderer requiring random number generation, quasi-Monte Carlo sampling, CDF inversions and recursion, involving both incoherent data access patterns and significant SIMD divergence.

Comparison to hand-written code. To better quantify the performance of IVL's code we also ran some experiments where we compared IVL-generated code to manually-written reference intrinsics code (in all cases, the reference code was written *before* the IVL code). For the `eyelight` ray tracer, IVL renders a 1600×1024 frame in 13.8 million cycles, vs 13.28m cycles in reference intrinsics code, a difference of only 4% (an exact reference version for the AO renderer is not available, due to that code's complexity).

Finally, we also ran an artificial *k-nearest neighbor* benchmark for which we had reference code for a variety of architectures. This benchmark is highly non-trivial in both control-flow and data access patterns. In this workload, IVL requires 755 million cycles (for 1 million 50-neighbor queries in a 1 million point dataset), as compared to 656m cycles for the hand-coded version—a difference of 15%.

[3] For example, IVL does not support doubles, yet, so both the IVL *and* the reference implementation have been modified to use floats.

The performance gap is mostly encumbered by that fact that IVL is a source-to-source compiler emitting vectorized code and relies on `icc` for optimizations. This compiler does not yet recognize some patterns which are suitable for optimizations but would have been applied by a human intrinsics programmer. `ispc` (see Section 6) on the other hand generates LLVM assembly code and ships its own set of specific optimizations for such patterns.

6. Related Work

While some efforts exist to integrate short vector types into languages, over the past years several data parallel languages have been developed in order to tackle programming for SIMD hardware. These languages can be basically subdivided into *data-parallel languages* and *vector* (or *array*) *programming languages*. Independent from that, several automatic vectorization techniques exist.

Short vector data types. With the help of C++ operator overloading or built-in support for short vector types in C [9, sec. 6.49], standard operator syntax can be used instead of writing intrinsics. This does not only increase readability but also portability of the source code as each back-end can resort to its specific SIMD ISA. However, type and control-flow vectorization must still be handled manually by the programmer. Especially, manual conversion of control-flow to data dependence is problematic as this—besides being very error-prone—again introduces architectural dependent low-level details to the source code: This approach has no means of abstracting blending/predicated execution.

Data-parallel languages. OpenCL and CUDA have already been discussed in Section 1. These languages are strongly influenced from shading languages whose programming model is similar but more limited than the former ones. One of the first shading languages implemented is RenderMan [10] which originally targeted a virtual SIMD array machine. Programs for this language are also written for a single scalar thread, while the compiler and the runtime system assured to instantiate this thread for all available SIMD lanes. RenderMan pioneered the concept of *uniform* and *varying* variables. *Uniform* variables are only set once for each set of properties. *Varying* variables on the other hand are allowed to change as a function of position. For instance, vertex normals in a Phong shader must be interpolated and must therefore be declared as *varying*.

Our work is close to the Intel® SPMD Program Compiler (`ispc`) [12]. This LLVM based compiler, where a program is instantiated n times to run on n SIMD lanes, currently supports several SSE revisions and AVX. Although IVL and `ispc` are two independent projects, we fostered an ongoing discussion with the `ispc` authors, which influenced each other's work. Thus, `ispc` and IVL/VecImp share many concepts: implicit handling of blending, automatic control-flow vectorization and **`uniform`** and **`varying`** variables among other features. There are also some differences: `ispc` does not support parametric polymorphism while `ispc`'s pointer support is more mature than IVL's one, for instance.

Another data-parallel language is C^n [22] which is in contrast to many other languages a real extension to C. C^n targets a special kind of SIMD machine, similar to IBM's Cell processor. There, a main processor controls n processing elements (PEs) that all have their own local memory. C^n extends the type system by two modifiers `mono` and `poly`. `poly` values sit in the memory of the PEs. The type system is used to trigger DMA transfers from the main CPU to the PEs. However, due to this different hardware model, C^n does not support `poly` values in compound data structures. More precisely, `mono` and `poly` are *storage-class specifiers* and not *type qualifiers*.

Vector programming languages. These languages are very powerful when dealing with mathematical vectors and matrices, i.e., arrays of atomic types. But this is also the main problem: Since arrays are a too high-level abstraction for short vector SIMD processors, it is very difficult to write efficient code for these machines when dealing with higher data structures. There does not exist any built-in way in order to deal efficiently with compound data, as this would contradict the array programming philosophy.

In some languages of this category like APL [15] or MatLab [23], the programmer has to manually vectorize control-flow. Others like Vector C [21], Vector Pascal [25], Intel's Array Building Blocks (ArBB [24], formerly known as Ct [8]) or Nesl [4] provide special constructs and/or enhance scalar constructs to work with vectors for this task.

Nesl integrates nested vectors as first class citizens in a pure functional language similar to ML. However, Nesl does not integrate imperative control structures and low-level constructs like pointers as we do.

ArBB/Ct is inspired by Nesl and can be seen as a Nesl library for C/C++. ArBB is a functional array language with an interface to C++. At first glance, it looks like a language extension to C++ for vector processing. However, ArBB uses operator overloading to construct an intermediate representation of the vector program which is compiled and executed *lazily* by a just-in-time compiler in the ArBB library. While this has several advantages, like adaption of vectorization to the system the program is currently running on, it has the disadvantage of having a second language within another language: ArBB requires own types (`i32` instead of `int`, etc.), own control structures (`_for ... _end_for` instead of the common `for` loop), and sometimes own operators (`((call)(f)(a, b)` instead of `f(a, b)`). Furthermore, the user has to understand the execution model precisely: ArBB expressions are used to *construct* a program. Putting `a+=b` in an *ordinary* loop will create as many expressions in the IR as the loop iterates. VecImp however combines both, scalar and vector computing in the same language.

Automatic vectorization. Vectorizing compilers go back to the early 1980s [1, 2]. Basic predication techniques have been introduced in this line of work [1]. Many different automatic vectorization techniques have been developed since then. The majority of approaches target vectorization of straight-line code loop bodies [28, 35]. There, instruction-level parallelism (ILP) is created by loop unrolling, or more sophisticated transformations (e.g. polyhedral techniques [5]) are applied. Other approaches exploit ILP in code that can contain control structures (like conditionals and loops) [18, 19, 33] for vectorization or vectorize a whole function [16]. Often, those approaches rely on preceding transformations that create this ILP. Outer-loop vectorization [26, 29, 31] goes beyond a loop body by integrating inner loops into vectorization. Another idea is to use the SIMD register file as fast cache [34].

As stated in the introduction, automatic vectorization is not applicable in our setting. The modifications to the data structure layout and the resulting changes in the code are not local to some code part. They often affect the whole program. Furthermore, algorithms substantially change, when they are programmed with SIMD instructions in mind. Current static analyses and code transformations are not able to perform such transformations automatically in a satisfactory way. Our work focuses on the programmer who explicitly wants to program for a CPU equipped with a SIMD instruction set without exposing her to assembly-level programming. This clientele is not satisfied with automatic techniques that only work on small code parts and depend on a preceding analysis that is not transparent to the programmer.

7. Conclusions and Future Work

The first practical experiences show that code emitted by IVL is comparable to hand-written intrinsic code. Smarter compiler transformations and back-ends would even produce higher quality code.

A hybrid SoA container class. As a next step, we would like to integrate this model into a C/C++ compiler. VecImp only changes

Imp where necessary and the added features are only minimum requirements in order to describe low-level SIMD programming in an abstract way. Often used programming patterns can be further abstracted with standard higher-level language features in order to reduce boilerplate code and increase portability. In particular, the C++ template mechanism would allow to define a hybrid SoA container class: Similar to `std::vector` which abstracts a traditional C array, one could implement a wrapper around a `T block[N]*`:

```
// scalar context throughout this example
struct vec3 { float x, y, z; };
// vec3 block[N]* pointing to ceil(n/N) elements
hsoa<vec3> vecs(n);
// preferred vector length of vec3 automatically derived
static const int N = hsoa<vec3>::vector_length;
int i = /*...*/
hsoa<vec3>::block_index ii = /*...*/
vec3 v = vecs[i];             // gather
vecs[i] = v;                  // scatter
vec3 block[N] w = vecs[ii];   // fetch whole block
hsoa<vec3>::ref r = vecs[i]; // get proxy to a scalar
r = v;                        // pipe through proxy
// for each element
vecs.foreach([](vec3& scalar v) { /*...*/ });
```

The inner type `hsoa<vec3>::ref` consists of a pointer pointing to the block in question and a lane index. Furthermore, this class is equipped with some conversion operators in order to automatically convert this proxy to the scalar base type (`vec3` in this case).

The `foreach` method is particularly interesting: Most of the time, i.e., while not processing one of the two border areas, each iteration can fetch a whole `vec3 block[N]& scalar`. The border areas must be handled specially—either by setting up an appropriate mask, or by scalarizing the border sections. This special setup is factored out in the `foreach` method which invokes different versions of the polymorphic lambda function and the user of `foreach` does not have to care about border areas anymore.

Block hierarchy. One could also imagine having a hierarchy of blocks. This information would also be annotated at the type and could be used in order to abstract blocks/work groups in CUDA/OpenCL or the layers of the cache hierarchy like cache lines and page sizes. A long term goal would be a language which emits SIMD CPU code and PTX/CUDA code dependent on the types which trigger the invocation of a kernel. Additional setup code would also be automatically generated.

Acknowledgments

The authors would like to thank Sigurd Schneider and Jan Schwinghammer who helped with the formalization of the type system. Furthermore, Ralf Karrenberg, Christoph Mallon and Phillip Slusallek were involved in the discussion of our work. William M. Mark, Matt Pharr and Sven Woop from Intel also gave a lot of valuable feedback and in particular the aforementioned collaboration regarding Matt's `ispc` project was crucial to the realization of IVL. Much of this work has been funded by the Intel Visual Computing Institute and the German ministry for education and research via the ECOUSS project.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *POPL*, 1983.

[2] R. Allen and K. Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Program. Lang. Syst.*, 1987.

[3] aobench. URL http://code.google.com/p/aobench/.

[4] G. E. Blelloch et al. Implementation of a Portable Nested Data-Parallel Language. In *PPOPP*, 1993.

[5] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhauser Boston, 2000.

[6] M. Farrar. Striped Smith–Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23:156–161, January 2007.

[7] I. Georgiev and P. Slusallek. RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. In *IEEE/Eurographics Symposium on Interactive Ray Tracing*, 2008.

[8] A. Ghuloum et al. Future-Proof Data Parallel Algorithms and Software on Intel Multi-Core Architecture. *Intel Technology Journal*, 11(04), November 2007.

[9] GNU Press. Using the GNU Compiler Collection. For GCC version 4.6.2.

[10] P. Hanrahan and J. Lawson. A Language for Shading and Lighting Calculations. In *SIGGRAPH*, 1990.

[11] Intel Corp. Intel® Compilers and Libraries. URL http://software.intel.com/en-us/articles/intel-compilers.

[12] Intel Corp. Intel SPMD Program Compiler. URL http://ispc.github.com.

[13] Intel Corp. Intel® 64 and IA-32 Architectures Optimization Reference Manual, 2009.

[14] Intel Corp. The Intel Many Integrated Core (MIC) Architecture, 2010.

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

[16] R. Karrenberg and S. Hack. Whole Function Vectorization. In *CGO*, 2011.

[17] Khronos Group. *OpenCL 1.0 Specification*, 2009.

[18] A. Krall and S. Lelait. Compilation Techniques for Multimedia Processors. *Int. J. Parallel Program.*, 28(4):347–361, 2000.

[19] S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. *PLDI*, 35(5):145–156, 2000.

[20] R. Leißa, S. Hack, and I. Wald. Extending a C-like Language for Portable SIMD Programming. The full version of our PPoPP'12 paper available online at http://www.cdl.uni-saarland.de/projects/vecimp.

[21] K.-C. Li and H. Schwetman. Vector C—A Vector Processing Language. *Journal of Parallel and Distributed Computing*, 2(2):132 – 169, 1985.

[22] A. Lokhmotov, B. R. Gaster, A. Mycroft, N. Hickey, and D. Stuttard. Revisiting SIMD Programming. In *LCPC*, pages 32–46, 2007.

[23] MatLab. URL http://www.mathworks.com/products/matlab.

[24] M. McCool. A Retargetable, Dynamic Compiler and Embedded language. In *CGO*, 2011.

[25] G. Michaelson and P. Cockshott. Vector Pascal, an array language, 2002.

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

[27] M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.

[28] D. Nuzman and R. Henderson. Multi-platform Auto-vectorization. In *CGO*, 2006.

[29] D. Nuzman and A. Zaks. Outer-Loop Vectorization: Revisited for Short SIMD Architectures. In *PACT*, 2008.

[30] NVIDIA. *CUDA Programming Guide*, 2009.

[31] R. G. Scarborough and H. G. Kolsky. A vectorizing Fortran compiler. *IBM J. Res. Dev.*, 30(2):163–171, 1986.

[32] L. Seiler et al. Larrabee: A Many-Core x86 Architecture for Visual Computing. In *SIGGRAPH*, 2008.

[33] J. Shin. Introducing Control Flow into Vectorized Code. In *PACT '07*, 2007.

[34] J. Shin, C. Jacqueline, and M. W. Hall. Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures. In *PACT*, 2002.

[35] N. Sreraman and R. Govindarajan. A Vectorizing Compiler for Multimedia Extensions. *Int. J. Parallel Program.*, 28(4):363–400, 2000.

[36] I. Wald. Fast Construction of SAH BVHs on the Intel® Many Integrated Core (MIC) Architecture. *IEEE Transactions on Visualization and Computer Graphics*, 99, 2010.

[37] J. Zhou and K. A. Ross. Implementing Database Operations Using SIMD Instructions. In *SIGMOD*, 2002.