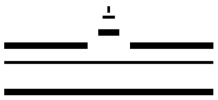


Westfälische Wilhelms-Universität Münster
Institut für Informatik

Diplomarbeit

Automatic SIMD Code Generation

Roland Leißa

March 5, 2010

Betreut durch Prof. Dr. Achim Clausing

Contents

Miscellaneous Lists

List of Abbreviations

ABI Application Binary Interface

API Application Programming Interface

AT&T American Telephone & Telegraph Corporation

AVX Advanced Vector Extensions

AoS Array of Structures

BSD Berkeley Software Distribution

CFG Control Flow Graph

CPU Computing Processing Unit

Cg C for Graphics

Def-Use Definition-Use

FPU Floating Point Unit

G++ GNU C++ Compiler

GCC GNU Compiler Collection

GLSL OpenGL Shading Language

GNU GNU's Not Unix[1]

GPU Graphics Processing Unit

HLSL High Level Shading Language

I/O Input/Output

[1] GNU is a recursive acronym.

ICC Intel C Compiler

IEEE Institute of Electrical and Electronics Engineers

IR Intermediate Representation

ISA Instruction Set Architecture

JIT Just In Time

MIMD Multiple Instructions, Multiple Data streams

MISD Multiple Instructions, Single Data stream

MMX Multimedia eXtension[2]

NaN Not a Number

OS Operating System

OpenCL Open Computing Language

OpenGL Open Graphics Library

PC Personal Computer

RAM Random Access Memory

SIMD Single Instruction, Multiple Data streams

SISD Single Instruction, Single Data stream

SSA Static Single Assignment

SSE Streaming SIMD Extensions

SoA Structure of Arrays

URL Uniform Resource Locator

VIS Visual Instruction Set

[2]Officially MMX is meaningless, but mostly it is referred as the stated abbreviation; other sources indicate the abbreviation stands for Multiple Math Extension, or Matrix Math Extension.

List of Algorithms

List of Figures

List of Listings

List of Tables

1 Introduction

SIMD instructions are common in microprocessors for roughly one and a half decade now. These instructions enable the programmer to simultaneously perform an operation on several values with a single instruction—hence the name: Single Instruction, Multiple Data. The more values can be computed simultaneously the better the speedup.

However, SIMD programming is still commonly considered as difficult. A typical programming approach which is still used today is presented by Chung [2009]: The programmer must usually deal with low-level details of the target machine which makes programming tedious and difficult. If such optimizations are used the resulting code is often not portable across other machines. This usually has the effect that two versions of an operation are implemented: One "normal" version used as fall-back which is programmed traditionally without the use of SIMD techniques and an optimized version which is used for a target machine of the programmer's choice. If other SIMD architectures should be supported more versions must be implemented. It is obvious that this makes testing and maintaining the source code complex.

More advanced SIMD programming approaches use uncommon data layouts. Thus it is difficult to put such an optimization in a late phase of a project which is nevertheless recommended as known from software engineering.[1]

Compiler researchers try to make automatic use of SIMD instructions. This is called *auto-vectorization* and is still an active research topic. Currently, it only works for relatively simple code pieces and even a perfect auto-vectorizer cannot change the data layout given by the programmer.

All this causes most programmers—even those working on multimedia projects which are prime examples for SIMD algorithms—to not bother with SIMD programming and the potential lies idle in many programs.

Instead of building a very smart auto-vectorizer a new programming paradigm is presented in this thesis which circumvents the problems of current SIMD programming techniques. Every high-level language which has support for records and arrays can principally be extended to this paradigm. We show by which constructs a language must be extended and what a compiler must do with them.

Picking up the idea of vector programming used in languages like APL, Matlab/Octave or Vector Pascal this programming technique is further enhanced so that such vector constructs even work for certain records. Additionally, operations defined for

[1] This is best described by Donald E. Knuth's famous quote: *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil* [cf. Knuth, 1979, pg. 228].

such records work in an SIMD context just as well and there is no need to implement the same functionality twice. This makes programming modular.

A compiler for a small experimental language called *Swift* featuring some of the constructs and algorithms proposed in this thesis has been implemented in order to give a proof of concept.

Chapter 2 defines the terminology and the intermediate representation (IR) used throughout this thesis. Furthermore, some compiler construction basics are given which are needed in later chapters. After a small history of SIMD computers Chapter 3 exemplarily and briefly discusses the x64 architecture and gives some general points on SIMD programming. The next chapter presents several data layouts suitable for SIMD programming while Chapter 5 discusses SIMD programming techniques which can be used today. The new paradigm is presented in detail in Chapter 6. The next chapter discusses its proof of concept implementation and some benchmarks. Chapter 8 gives some hints about further research topics and some ideas of getting rid of some restrictions in the algorithms introduced in Chapter 6. Finally, the last chapter briefly discusses all results and concludes this thesis.

2 Foundations

This chapter defines the terminology and the formal foundations used throughout this thesis. Furthermore, the algorithms presented in this thesis work on an abstract IR of a program in Static Single Assignment (SSA) Form. The following sections clarify what this means.

2.1 Terminology

2.1.1 Interpreters vs. Compilers

Although the term *compiler* is used throughout this thesis and a traditional compiler is meant which translates a high-level language to machine language, assembly language or object code, the algorithms presented in this place can be used with few or no changes in interpreters, bytecode compilers, virtual machines etc. where applicable.

2.1.2 Data Types

Since literature and programming languages are not consistent in the naming of data types the following terms are used throughout this thesis:

Byte: An 8 bit wide signed or unsigned integer.

Word: A 16 bit wide signed or unsigned integer.

Doubleword: A 32 bit wide signed or unsigned integer.

Quadword: A 64 bit wide signed or unsigned integer.

Single: A 32 bit wide IEEE-754 single precision floating point type.

Double: A 64 bit wide IEEE-754 double precision floating point type.

As long as it is not important whether an integer type is signed or unsigned nothing more is said about it. If it is important, however, it is referred as *signed* or *unsigned*, respectively.

The term *integer* is used as either a *byte*, *word*, *doubleword* or *quadword* either *signed* or *unsigned*. If this is important the type is referred as *signed* or *unsigned integer*. The set of all *integer* types is referred as T_{int}.

The term *float* is used as either a *single* or a *double*: $T_{float} \coloneqq \{single,\ double\}$.

A *boolean* is a 8 bit wide area. A value of 00_{16} represents *false* whereas any other value is interpreted as *true*.

Floats, *integers* and *booleans* are also called *base types*: $T_{base} \coloneqq T_{int} \cup T_{float} \cup \{boolean\}$.

A vector of n elements of the same *base type* T where n is a constant power of two is also called n *packed Ts*. All possible types are referred with the set T_{packed}. Thus an *8 packed unsigned word* means a 16 byte wide area with eight 16 bit wide *unsigned integers*. The elements are assumed to be side by side in memory. Hence the size of an n *packed* T is n times the size of T. Note that this also means that the size of a *packed type* in bytes is also guaranteed to be a power of two.[1]

The function num-elems-of : $T_{packed} \to \{n | n = 2^m, m \in \mathbb{N}\}$ is a map for which holds:

$$\text{num-elems-of}(n\ packed\ t) = n\ .$$

Pointers capture an exceptional position: Sometimes they can just be handled as an appropriate integer type, sometimes they are handled completely differently. Therefore a *pointer* is neither an *integer* nor a *base type*. The size of a *pointer* is an at compile time known size depending on the target machine. On a typical 32 bit machine the size would be four bytes whereas on a typical 64 bit machine the size would be eight bytes.

A *record type* with n attributes is a list $\mathcal{L}_n$ of n members with $n \in \mathbb{N}^{\neq 0}$ and each member is a tuple $(id,\ t)$ where id is a unique identifier and $t \in T_{all}$.

All possible *record types* are referred as the set T_{record} whereas all introduced types are contained in the set $T_{all} \coloneqq T_{base} \cup pointer \cup T_{packed} \cup T_{record}$.

The function size-of : $T_{all} \to \mathbb{N}$ is a map returning the size in bytes of the input type as defined in this section. Note that this function can be evaluated at compile time in all cases.

A Note on Arrays

Arrays usually come in two flavors: An array with an at compile time known constant size and an array with an at compile time unknown but constant size. The first one is just a special case of a *record type* where all members have the same type. The second one is almost always created dynamically on the heap and *pointers* are used to access the elements. In this thesis it is not necessary to have a special array type as special instructions are used to access the elements via the *pointer*. Therefore the algorithms presented here need not distinguish between array types and record types or array types and *pointers*, respectively.

However, when mentioning high-level constructs arrays are of course considered. Only the low-level IR of a program does not have to regard them as special type.

[1] Principally, *base types*, which are not a power of two, and/or *packed types*, which are a vector of n elements where n is not a power of two neither, are possible, but in practice these are considerations which simply do not occur because they are impractical.

2.2 Bit Arithmetic

As especially *packed types* are of greater interest for SIMD programming and the size of them is always a power of two, fast bit arithmetic can sometimes be useful instead of using slow integer arithmetic when dealing with an $n = 2^m$:

- A common programming trick is to use bit shifts instead of multiplications or divisions with a power of two:

  $a \leftarrow a << m \quad \triangleright$ instead of $a \leftarrow a \times n$
  $a \leftarrow a >> m \quad \triangleright$ instead of $a \leftarrow a \div n$

  The free slots after a left-shift must simply be filled with zeros while a right-shift must take care whether a *signed* or an *unsigned integer* gets shifted. In the case of an *unsigned* shift where the most significant bit is set, the free slots must be filled with ones, while in all other cases they must be filled with zeros.

- Modulo can be simulated with an appropriate bitwise AND:

  $a \leftarrow a \textbf{ and } 0 \ldots 0 \underbrace{1 \ldots 1}_{m \text{ times}}{}_2 = a \textbf{ and } (n-1) \quad \triangleright$ instead of $a \leftarrow a \bmod n$

- When using the negated mask the operation gets the previous multiple of n if a is not currently a multiple of n:

  $a \leftarrow a \textbf{ and } 1 \ldots 1 \underbrace{0 \ldots 0}_{m \text{ times}}{}_2 = a \textbf{ and not } (n-1) \quad \triangleright$ instead of $a \leftarrow a \bmod n$

- In order to get the next multiple of n, if a is not already a multiple of n, the following calculation can be performed:

  $a \leftarrow (a + n - 1) \textbf{ and } 1 \ldots 1 \underbrace{0 \ldots 0}_{m \text{ times}}{}_2 = (a + n - 1) \textbf{ and not } (n-1)$

2.3 Programs

In this thesis it is only necessary to look at one procedure at a time. Traditionally this unit is called a *program*. A program consists of a list of instructions. Each instruction takes some arguments in the form of constants or variables and computes some results in the form of variables.

A variable is a tuple $(value, id, t)$ where $value \in \mathcal{D}$, id is a unique identifier and $t \in T_{all}$. The set $\mathcal{D}$ denotes the set of all possible values.[2] A constant is a tuple (c, t) where $c \in \mathcal{D}$. The set of all variables used in a program P is called V_P while the set of all used constants is called C_P. We define the set of operands used in a program P as $O_P \coloneqq V_P \cup C_P$.

[2] We do not define this set exactly as a precise definition serves no purpose in this thesis.

The function type-of : $O_P \to T_{all}$ returns for a given variable or constant its corresponding type.

The map value-of : $O_P \to \mathcal{D}$ returns for a given operand its value. As writing value-of(v) for some variable v is tedious we write abbreviated just v for convenience. Similarly, we often just write 4 for a constant of some appropriate type which holds the constant of value 4. Furthermore, we use square brackets to index the base values of a *packed type* $t \in T_{packed}$. The first element has the index 0 while the last element has the index $n-1$ with $n =$ num-elems-of(t).

In many examples we only need *base types* and *packed types*. For convenience we often need not distinguish between the exact types there. Variables and constants of *base types* simply appear in normal print whereas operands of *packed types* appear with a small arrow above. A single constant with an arrow means that the vector has this value at all elements.

2.4 Instructions

The exact semantics of different instructions are only loosely defined because different IRs in compilers may use different instruction sets and the examples should be self-explanatory. However, instructions with *packed types* need clarification.

Typical unary and binary operations like elementary arithmetic, logical operations, unary minus and unary NOT with *packed types* use the same semantics as the scalar version as if the operation would simply be executed for each element of the variable in a parallel way. We distinguish a scalar operation from a packed operation by putting a circle around the operation. For instance, consider three vector variables $\vec{a} = (a_0,\ a_1,\ a_2,\ a_3)$, $\vec{b} = (b_0,\ b_1,\ b_2,\ b_3)$ and $\vec{c}$. After execution of an SIMD addition

$$\vec{c} \leftarrow \vec{a} \oplus \vec{b}$$

the variable $\vec{c}$ yields

$$\vec{c} = \begin{pmatrix} a_0 + b_0 \\ a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{pmatrix} =: \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} .$$

Figure 2.1 illustrates this computation graphically. With $\vec{a} = (1.0, 2.0, 3.0, 4.0)$ and $\vec{b} = (2.0, 4.0, 8.0, 10.0)$ it follows that $\vec{c}$ will be evaluated to $(3.0, 6.0, 11.0, 14.0)$.

Comparisons are a special case. A comparison of two n *packed Ts* of the same size and same number of elements results in an appropriate n *packed unsigned integer* of the same size and number of elements instead of a *boolean*. Each field of the result represents a bit mask. A bit mask of all ones represents *true* whereas a result of all zeros represents *false*. Other values are not allowed and cause undefined behavior.

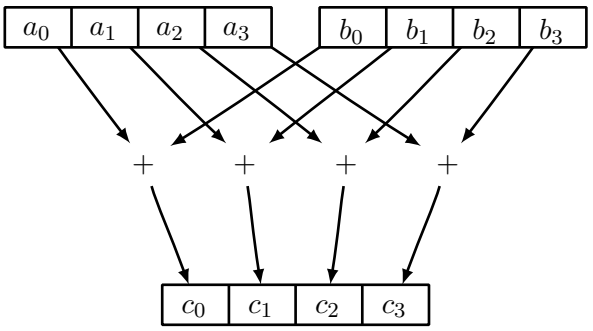


Figure 2.1: An SIMD addition of four elements

Consider the following operation where $\vec{b} = (1.0, 2.0, 3.0, 4.0)$ and $\vec{3}$ are *4 packed singles*:

$$\vec{a} \leftarrow \vec{b} \circledcirc \vec{3} \ .$$

Then $\vec{a}$ is *4 packed unsigned doubleword* and is evaluated to

$$(\mathrm{FFFFFFFF}_{16}, \mathrm{FFFFFFFF}_{16}, 00000000_{16}, 00000000_{16}) \ .$$

For the algorithms which are presented in Chapter 6 it is necessary to define which instructions modify the control flow of the program. In most cases, after the evaluation of an instruction during execution, simply the next instruction in the list will be executed. Special *jump instructions* may change this behavior. These instructions take *label instructions* as additional arguments which are the possible control flow targets after the evaluation of a *jump instruction*. *Label instructions* have a unique identifier *id* and just serve as jump targets and do not have any other side effects. We prefix a label with an l and use the identifier as index. Thus we write

$$l_{id} :$$

and the control flow continues linearly from there on as normal. Furthermore every program's instruction list must start and end with a (distinct) label.

A *jump instruction* serves as an umbrella term for two instructions. The first one is called *goto instruction* which just takes one label l_{id} as argument:

$$\textbf{goto}\, l_{id} \ .$$

After evaluation the execution continues at the given target label.

The latter instruction is the *branch instruction*. It expects an operand $c \in O_P$ with type-of$(c) \in \{boolean\} \cup T_{pui}$ where T_{pui} is the set of all *packed unsigned integers* and does not have any results. Furthermore it expects two target labels. We simply write

$$\textbf{if}\ c\ \textbf{then}\ l_{true}\ \textbf{else}\ l_{false} \ ,$$

$a \leftarrow \ldots$
if $a > 3$ **then**
    $a \leftarrow 1$
else
    $a \leftarrow 2$
end if
$\cdots \leftarrow a$

Figure 2.2: High-level if-else-construct

if type-of$(c) = \mathit{boolean}$. In this case the execution continues at l_{false}, if c evaluates to *false*, and it continues at l_{true} otherwise. We write

$$\textbf{if } \vec{c} \textbf{ then } l_{\mathit{true}} \textbf{ else } l_{\mathit{false}} \ ,$$

if type-of$(\vec{c}) \in T_{\mathit{pui}}$. The execution only continues at l_{false}, if all elements of $\vec{c}$ evaluate to *false*, otherwise it continues at l_{true}. In other words the execution only continues at l_{false}, if $\vec{c}$ is completely filled with zeros.

Furthermore, we require that every jump instruction is followed by a label. This is important to build basic blocks (see next section).

These definitions may be changed or extended but this will most likely result in a change of the vectorization algorithms.

The pseudo-code in Figure 2.2 shows a typical if-else-statement similar to many high-level languages. Figure 2.3a shows the most obvious variant of this program in the above defined IR.

2.5 Basic Blocks and the Control Flow Graph

A program can be divided into *basic blocks*. A sublist of a program's instruction list is a basic block if it fulfills three conditions:

1. The first instruction in the sublist is a label.
2. No other label is in the sublist.
3. Only the last instruction in the sublist may be a *jump instruction*.

Note, that neither a jump can be made directly into the middle of a block nor is there a jump out of the middle of a block. The basic block is identified via its leading *label*. In literature basic blocks are usually defined differently but this definition works better with the above defined IR.

l_{start}:
  $a \leftarrow \ldots$
  $c \leftarrow a > 3$
  if c **then** l_{true} **else** l_{false}
l_{true}:
  $a \leftarrow 1$
  goto l_{next}
l_{false}:
  $a \leftarrow 2$
l_{next}:

  $\ldots \leftarrow a$

(a) IR of the program in Figure 2.2

l_{start}:
  $a_1 \leftarrow \ldots$
  $c \leftarrow a_1 > 3$
  if c **then** l_{true} **else** l_{false}
l_{true}:
  $a_2 \leftarrow 1$
  goto l_{next}
l_{false}:
  $a_3 \leftarrow 2$
l_{next}:
  $a_4 \leftarrow \Phi(a_2, a_3)$
  $\ldots \leftarrow a_4$

(b) IR in SSA form of the program in Figure 2.2

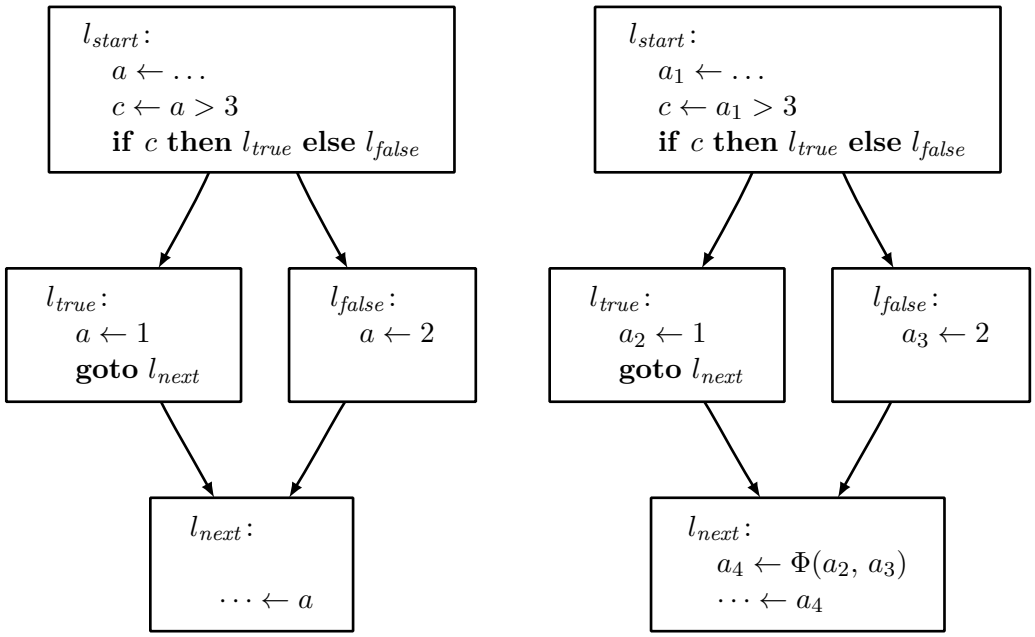
Figure 2.3: IR of an if-else-statement

The basic blocks and the control flow can be visualized with a *control flow graph* (CFG). The CFG is a directed graph where each basic block is a node in the graph and an edge is added for each possible control flow from block to block. Figure 2.4a shows how the sample program of Figure 2.3a can be visualized. Each basic block b has a set of predecessor basic blocks referred as $pred(b)$ and a set of successor basic blocks denoted by $succ(b)$.

We say a program with the starting label *start* is *strict* if for each variable v and for each usage of v at instruction i holds: Each path from *start* to i contains a definition of v.

A non-strict program can be transformed into a strict one by inserting a dummy definition at the start of the program (after the start label) to a dummy value `undef` for each variable which violates the strictness property. No code must be generated for this assignment, of course. From now on we only consider strict programs.

We say a block b_1 *dominates* b_2 in a program with a starting block b_{start} if and only if each path $b_{start} \rightarrow^* b_2$ contains b_1 and we shortly write $b_1 \preceq b_2$. Note that this definition implies that each basic block dominates itself. Therefore we say a block b_1 *strictly dominates* b_2 if b_1 dominates b_2 and $b_1 \neq b_2$. We shortly write $b_1 \prec b_2$.



(a) CFG of the program in Figure 2.3a (b) CFG of the program in Figure 2.3b

Figure 2.4: The CFG of a program and its corresponding program in SSA form

The term of dominance can be extended to instruction relations, too: An instruction i_1 in basic block b_1 *dominates* i_2 in block b_2 if $b_1 \prec b_2$ or if $b_1 = b_2$ and i_2 succeeds i_1 or $i_1 = i_2$. An instruction i_1 *strictly dominates* i_2 if i_1 dominates i_2 and $i_1 \neq i_2$. We also write $i_1 \preceq i_2$ or $i_1 \prec i_2$, respectively.

Furthermore, we define the dominance frontier of a block b:

$$df(b) = \{n \mid b \preceq p \in pred(n) \wedge b \not\prec n\}$$

It is possible to build a CFG which is not connected if the source program consists of a trivial infinite loop which is followed by other instructions (which will never be executed) as shown in Figure 2.5. Basic blocks which are unreachable from the starting block can be safely removed as these blocks emerge from dead code. From now on we always assume a connected CFG. If such problems are eliminated the starting basic block of a program dominates all basic blocks of the program.

Since the dominance relation is transitive[3] a tree order is induced on the basic blocks where all siblings share a unique dominator called *immediate dominator*. We denote the

[3] In fact it is even an order relation as it is reflexive and anti-symmetric, too [see Lengauer and Tarjan, 1979].

l_{loop}:
    ...
    goto l_{loop}
l_{next}:
    ... ▷ This will never be executed

Figure 2.5: Trivial infinite loop

immediate dominator of a basic block b by $idom(b)$. Figure 2.6a shows the CFG of a more complex program. Figure 2.6b shows the corresponding dominator tree.

More information how the dominance relation can be computed is described by Lengauer and Tarjan [1979] and Cooper et al. [2001], for instance.

2.5.1 Critical Edge Elimination

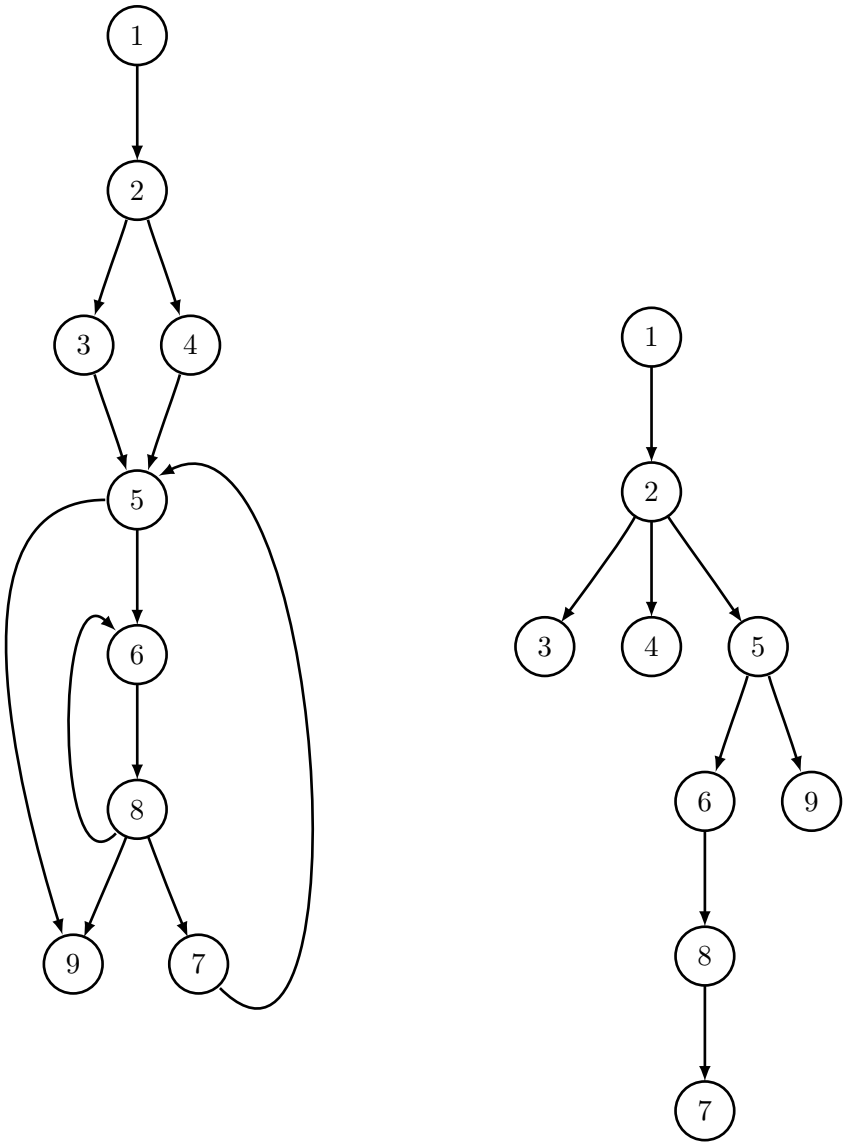
An edge $i \to j$ is called *critical* if $|pred(j)| > 1$ and there exists a node $k \neq j$ with $i \to k$. Many algorithms require that such edges are eliminated. This can be achieved by inserting a new empty basic block as shown in Figure 2.7.

2.6 Static Single Assignment Form

A very popular program transformation in compilers is a conversion to SSA form. In this form every variable is defined exactly once. Compared to traditional IRs this bears many advantages. Some of them among others are:

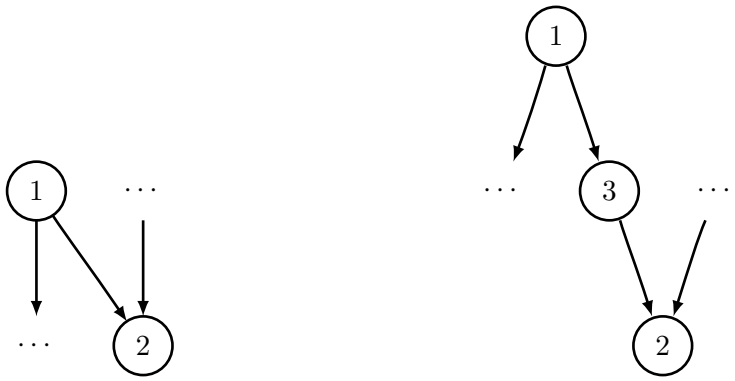
- Simple data flow information is normally kept in def-use chains. A variable with d definitions and u uses needs $d \times u$ def-use chains. In SSA form each variable exactly has one definition with u uses.
- Many optimizations like dead code elimination and constant propagation become much easier if every variable is defined exactly once.
- The inserted Φ-functions (see below) yield very useful information for many optimizations which are also used in the vectorization algorithms in Chapter 6.
- Theorem 2.1 is very useful for many algorithms.

The transformation to SSA form is done by renaming each definition of the same variable and substitute its uses accordingly. However, this yields the problem that a new definition of a variable which depends on two other definitions of the same variable in different predecessor basic blocks does not know which one is the proper one to select. Figure 2.4 illustrates the problem: In the l_{next} block the redefinition of a depends on



(a) CFG of a more complex programm (b) Dominator tree of the programm

Figure 2.6: A CFG of a program and its corresponding dominator tree



(a) The edge $1 \rightarrow 2$ is critical.

(b) The insertion of a new empty basic block 3 eliminates the critical edge.

Figure 2.7: Critical edge elimination

the definition of a in the l_{true} block and on the definition in the l_{false} block (see Figure 2.4a). Therefore so-called Φ-functions are introduced which select the proper variable (see Figure 2.4b).

We must often distinguish between Φ-functions and other instructions. We refer all instructions which are not Φ-functions as *ordinary* instructions.

Φ-functions—if present—are the first instructions in a basic block after the leading label instruction. Each Φ-function *phi* exactly has one result and takes n arguments $\in V_P$ where n is the number of predecessor basic blocks. Each argument v corresponds to exactly one predecessor basic block p. We express this by writing $pred(phi, v) = p$.

The implementation must ensure that a copy is made which copies the source location of the variable in question (the appropriate argument of the Φ-function) to the target location (the result of the Φ-function) during the execution of a control flow edge. The location may be a register or a memory slot depending on how the variable is handled at that moment. Of course, no copies are necessary if both locations are equal i.e., the same register or the same memory slot.[4]

This also shows how Φ-functions must be handled when considering liveness information and the like:[5] Each argument of a Φ-function increases the lifetime of this variable to the end of its corresponding preceding basic block but not further. In other words this means with an argument v of a Φ-function *phi* in basic block b, that v is live-in and live-out at $pred(phi, v)$'s last instruction. But v is neither live-out in $pred(phi, v)$ nor live-in in b nor is it in *phi*.

[4] An optimization pass which tries to merge as many locations as possible is called *coalescing*.

[5] A liveness definition of a program in SSA form is given by Hack [2007, sec. 4.1], for instance. Appel [1998, sec. 19.6] presents an algorithm in order to perform liveness analyses for programs in SSA form.

Furthermore this means for a program to be strict that it suffices that the definition d_v of a variable v only needs to dominate $pred(phi, v)$'s last instruction of a use in a Φ-function phi.

We can express the strictness property of a program in SSA form by the following definition:

Definition 2.1 (SSA Strictness)**.** *A program in SSA form with a starting label start is called strict if for each variable v and for each usage at an ordinary instruction i holds: Each path from start to i contains the definition of v. For each variable v and for each usage at a Φ-function phi must hold: Each path from start to $pred(phi, i)$'s last instruction contains the definition of v.*

We can express this in other words by the following theorem:

Theorem 2.1 (SSA Strictness Property)**.** *Let P be a strict program in SSA form and $v \in V_P$ be a variable defined at instruction d_v. For each use of v in an ordinary instruction i holds: $d_v \prec i$. For each use of v in a Φ-function phi where l is the last instruction of $pred(phi, v)$ holds: $d_v \preceq l$.*

Proof. The theorem directly follows from Definition 2.1 □

Lemma 2.1. *Let P be a strict program in SSA form with n instructions. Every instruction defines at most a constant number of variables and takes at most a constant number of constant arguments. Now it holds:*

$$\exists c \in \mathbb{N} : v + k \coloneqq |V_P| + |C_P| = |O_P| < cn \ .$$

Remark. This lemma is needed to make running time estimations only dependent on the number of instructions. The postulation that every instruction defines at most a constant number of variables and only takes at most a constant number of constant arguments is not necessarily realistic. But this consideration justifies this approach: If we push these constants to a high value, which is not virtually exceeded, everything is fine. If this value is exceeded anyway, we have to split the instruction in question which makes the program slightly longer. In practice it has been observed that in average about one to two variables are defined per instruction and every instruction does not usually take more than three constant arguments. For some IRs—three-address code, e.g., as the name suggests—this assumption holds, however.

Proof. Let n_v be the maximum number of variables defined and n_c the maximum number of constant arguments per instruction. Since the program is strict, it holds:

$$v + k < n_v n + n_c n = (n_v + n_c)n =: cn \ .$$

□

Figure 2.3b shows the sample program of Figure 2.3a in SSA form. The corresponding CFG can be seen in Figure 2.4b.

A more detailed introduction to SSA form is given by Cytron et al. [1991]. A constant propagation algorithm, a dead code elimination technique and a combined approach called *sparse conditional constant propagation* is presented by Click and Cooper [1995]. These algorithms make all use of SSA form. Hack [2007] describes a register allocator directly based on SSA form.

3 The SIMD Architecture

Flynn's taxonomy [see Flynn, 1972] suggests a classification of different computer architectures. These are:

SISD (Single Instruction, Single Data stream): This is the classic programming model where a single instruction operates on a single datum.

SIMD (Single Instruction, Multiple Data streams): In an SIMD environment a single instruction can operate on a data stream with several elements. Usually, a single instruction fetches a number of elements of the stream, which is a power of two, while all elements lie side by side in memory.

MISD (Multiple Instructions, Single Data streams): Several systems operate simultaneously on a single data stream in this very uncommon model. In the end these systems must usually agree on a result.

MIMD (Multiple Instructions, Multiple Data streams): Here several autonomous systems operate on several data streams. The data may be shared between the systems or lie in an distributed environment.

Some even distinguish more models or subdivide the existing ones. But in this thesis only the SISD and the SIMD model are of further interest. However, it should be outlined that even MIMD computers may consist of SIMD nodes and no further problems occur when using SIMD instructions within a multiprocessor environment. When speaking of computations, instructions, etc. in an SISD context we use the adjective *scalar* whereas we sometimes use the phrase *vector computations*, *vector instructions* etc. when meaning SIMD ones.

3.1 A Brief History of SIMD Computers

In the 1980s SIMD architectures were popular among super computers. When the MIMD approach became less expensive interest in SIMD machines decreased in favor of MIMD systems. In the early 1990s Personal Computers (PCs) became common in private

households and replaced home computers.[1] PCs were capable of 3-D real-time gaming[2] which increased the interest for SIMD computing again as a cheap alternative for more computing power.

With the introduction of the MMX Instruction Set Architecture (ISA) as an extension to the x86 architecture in 1996, SIMD programming became widely deployed as this architecture was (and still is today) by far the most used processor family in private computers. Other microprocessor vendors also added SIMD extensions to their chips: Sun had already introduced the Visual Instruction Set (VIS) enhancement to their SPARC processors in 1995 while IBM and Motorola enhanced their POWER chips with the AltiVec extension. Over the years the x86 and the POWER architectures gained several SIMD extensions while other microprocessors like the Cell processor picked up SIMD computations, too.

Intel and AMD are currently working on a new generation of SIMD instructions. This ISA will be called Advanced Vector Extensions (AVX).[3] It will introduce completely new 256 bit wide registers for use with appropriate *packed float* and *integer* types. Additionally, those instructions will allow three operands instead of only two which was the case for almost all other prior instructions. As one can see the developing of multi-core processors did not stop the research of new SIMD technologies.

3.2 The x64 Architecture

Intel and AMD started work on 64 bit processors as it was only a question of time till the $2^{32} = 4\,\text{GiB}$ address space of 32 bit processors would be exceeded. While Intel developed the Itanium processor—a completely new architecture—AMD designed their 64 bit processor very similar to the x86 architecture. This was called *AMD64*. As the Itanium processor was economically a big failure Intel had to copy AMD's design.[4] They called this architecture *Extended Memory 64 Technology* (EM64T) and renamed it later on to *Intel 64*. Because both architectures are very similar from a programmer's point of view even Intel 64 is today frequently referred to as AMD64—often for historical

[1] With the term *home computer* a special class of personal computers is meant which became increasingly popular in private households during the late 1970s and the whole 1980s. These machines were typically less expensive than business machines and provided therefore less memory and computing power. Because these computers often served as gaming stations the graphic and sound capabilities were surprisingly comparable or even better than those of business machines. Typical exponents of this class of computers were several Atari and Commodore machines, for instance.

[2] The game *Doom* released by *id Software* in 1993 is commonly considered as milestone in 3-D gaming history pioneering sophisticated 3-D graphics on mainstream hardware at that time.

[3] See http://software.intel.com/en-us/avx/ and http://forums.amd.com/devblog/blogpost.cfm?threadid=112934&catid=208.

[4] See http://news.zdnet.com/2100-9584_22-145923.html.

reasons.[5] The generic term is *x86-64*[6] which is often abbreviated with *x64*. The latter term is used in this thesis.

Since the proof of concept implementation, which is presented in Section 7.1, uses the x64 architecture as back-end this architecture is exemplarily taken to discuss an SIMD processor.

As already said MMX was the first technology on x86 which implemented SIMD instructions. The associated instruction set only supports integer arithmetic. Eight 64 bit wide registers are available—also known as MMX registers—which reuse parts of the floating point registers of the Floating Point Unit (FPU). As especially floating point arithmetic is important for gaming further instruction sets were developed. AMD developed the 3D-Now! extension which uses the same registers and allows calculations with *2 packed singles* while Intel's SSE introduces eight completely new 128 bit wide registers—called XMM registers—for use with *4 packed singles* and corresponding *packed integers*. SSE2—also developed by Intel—introduces among other instructions support for *2 packed doubles*. Further SSE enhancements (SSE3 and SSE4) introduced only instructions for special calculations. Since larger registers must be regarded as superior, AMD gradually copied Intel's SSE enhancements calling it *128-Bit Media Instructions* as SSE is a trademark by Intel.

SSE and SSE2 capable processors can deal with the following data types when using XMM registers:

- 4 packed singles
- 2 packed doubles
- 16 packed signed and unsigned bytes
- 8 packed signed and unsigned words
- 4 packed signed and unsigned doublewords
- 2 packed signed and unsigned quadwords

Furthermore so-called *scalar* versions of the float instructions are available which operate on a single float. In this case the remaining part of the register is simply unused.

The alignment of the used data is very important. Data which are aligned on a 16 byte boundary can be fetched/stored with a fast aligned move (**movaps**—Move Aligned Packed Singles, for example) whereas data with improper alignment or where the alignment is uncertain at compile time must be fetched/stored with a slower unaligned move

[5] See http://www.gentoo.org/proj/en/base/amd64/, for example.
[6] The terms *X86-64*, *x86_64* and *X86_64* are also common.

```
movaps    (%rsi), %xmm0 // xmm0 = *rsi
movaps 16(%rsi), %xmm1 // xmm1 = *(rsi + 16)
addps     %xmm1, %xmm0 // xmm0 *= xmm1
movaps    %xmm0, (%rsi) // *rsi = xmm0
```

Listing 3.1: Adding up two *4 packed singles*

(**movups**—Move Unaligned Packed Singles, for instance).[7] An aligned move with unaligned data results in a segmentation fault.

The x64 processor can run in several modes. If run in *64-Bit Mode* which also implies an 64 bit Operating System (OS) almost all instructions known from the x86 processor are available, too. The general purpose registers are enhanced to be 64 bit wide and the instruction set is adapted properly. The number of general purpose registers and the number of available XMM registers are doubled in 64-Bit Mode resulting in 16 registers each.

All x64 processors are capable of the instruction set introduced with SSE and SSE2. In order to make programming examples as consistent as possible this architecture in 64-Bit Mode is assumed. Furthermore, assembly language examples are given in AT&T syntax instead of Intel syntax because the former one is known to be less verbose while still being more precise than the latter one.[8]

For a more detailed introduction to this architecture refer to AMD [a, chap. 1] or Intel [b, chap. 2].

3.3 Simple SIMD Programming

We have already discussed SIMD instructions of the IR defined in Section 2.4. Simple tasks like adding up two vectors from memory are easily coded as shown in Listing 3.1: The register %**rsi** points to two *4 packed singles* in memory. It is assumed that this address is dividable by 16. Thus these values can be fetched with an aligned move and added up. The result is written back to the location pointed to by %**rsi**.

3.3.1 Elimination of Branches

SIMD Programming becomes more difficult if branches are involved:

for $i = 0$ to 3 **do**
  if $(a + i) < 3$ **then**

[7] Refer to Intel [c] and Intel [d] or AMD [b], AMD [c] and AMD [d] in order to lookup the semantics of assembly language instructions here and in following examples.

[8] See for differences between these syntaxes: http://www.redhat.com/docs/manuals/enterprise/RHEL-3-Manual/gnu-assembler/i386-syntax.html.

$\vec{v}[i] \leftarrow i$
end if
end for

Only in the case that $a + i$ is less than 3, i should be written into the i^{th} element of $\vec{v}$. If we assume—contrary to the definition in Section 2.4—that a comparison of a vector yields 1 for all elements for which the condition is true and 0 otherwise, the if-else-statement can be eliminated by writing:

$\vec{i} \leftarrow (0, 1, 2, 3)$
$\vec{c} \leftarrow (a + \vec{i}) \lessdot 3$
$\vec{n} \leftarrow$ **not** $\vec{c}$ ▷ means: 0 is converted to 1 and vice versa
$\vec{v} \leftarrow i \odot \vec{c} \oplus \vec{v} \odot \vec{n}$

This style of programming is recommended in Matlab/Octave, for instance.[9] If we consider the binary representation of the data types we can also use bit arithmetic in order to get rid of the if-else-statement:

$\vec{i} \leftarrow (0, 1, 2, 3)$
$\vec{c} \leftarrow (a + \vec{i}) \lessdot 3$ ▷ this is a "normal" comparison again as defined in Section 2.4
$\vec{n} \leftarrow$ **not** $\vec{c}$
$\vec{v} \leftarrow (i$ **and** $\vec{c})$ **or** $(\vec{v}$ **and** $\vec{n})$

SSE even provides special instructions (**andnps**—And Not Packed Singles, for instance) which negate the second operand and then perform a bitwise logical AND[10] [see AMD, c, pg. 17–20 and pg. 242–243]. A proper mask is usually generated using a special comparison (**cmpps**—Compare Packed Singles, for example) [see AMD, c, pg. 26–31 and pg. 248–259]. This basically works like the comparison defined in Section 2.4. Thus the following pseudo-code can be optimized as shown in Listing 3.2:

for $i = 0$ to 3 **do**
if $\vec{v}_1 == \vec{v}_2$ **then**
$\vec{r}[i] \leftarrow \vec{a}[i]$
else
$\vec{r}[i] \leftarrow \vec{b}[i]$
end if
end for

Note that proper masking only takes three instructions. This demand can slightly increase if the values of some registers which gets destroyed in the process must be saved.

Alternatively, *blend* instructions provided by newer Intel CPUs can be used which basically do this masking in one instruction ([see Intel, c, pg. 3–75 – 3–86] and [Intel, d, pg. 4–71 – 4–77]). We just use Algorithm 3.1 for this masking task. If the target archi-

[9] See also Section 5.5.
[10] This operation should not be confused with a bitwise logical NAND.

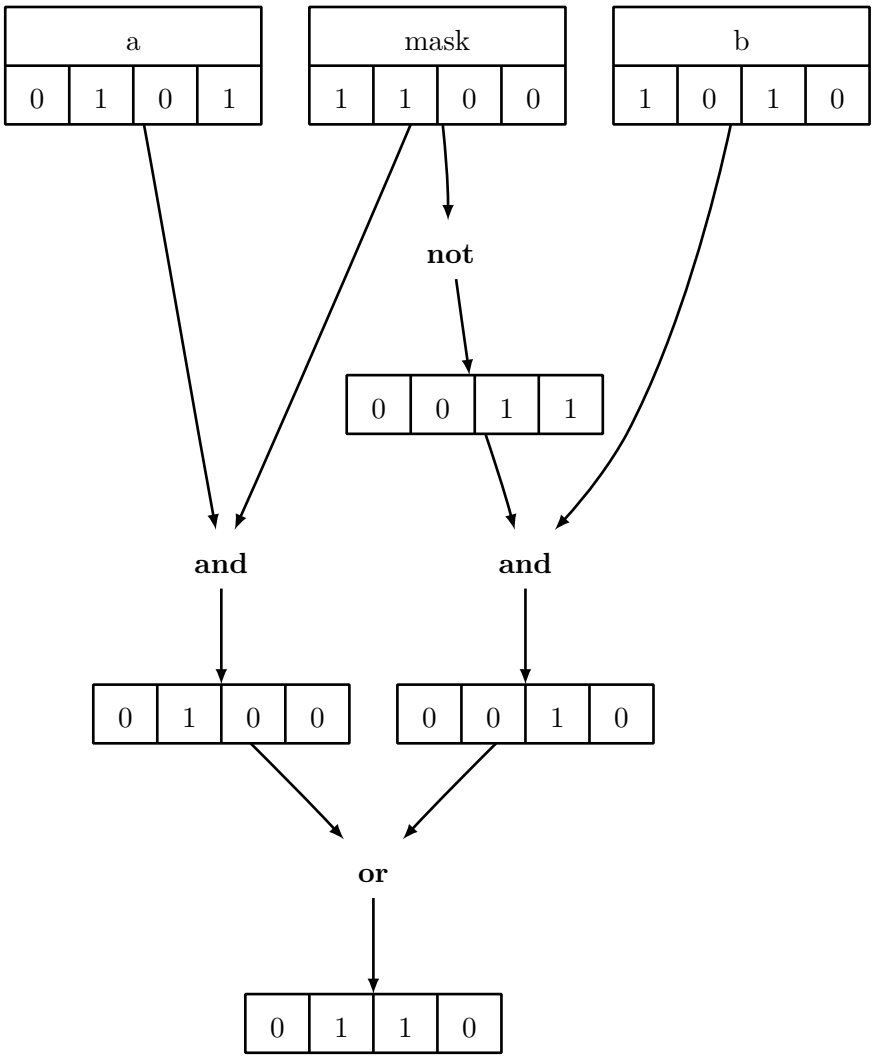


Figure 3.1: Masking different results and combining to one result

```
cmpeqps %xmm3, %xmm0 // mask(xmm0) = v1(xmm0) == v2(xmm3)
andps   %xmm0, %xmm1 // t(xmm1) = a(xmm1) AND mask(xmm0)
andnps  %xmm2, %xmm0 // f(xmm0) = b(xmm2) AND (NOT mask(xmm0))
orps    %xmm1, %xmm0 // r(xmm0) = t(xmm1) OR f(xmm0)
```

Listing 3.2: Blending two *4 packed singles* into one

tecture supports more sophisticated instructions the implementation may look different, of course.

Algorithm 3.1 BLEND($\vec{c}$, $\vec{o}_1$, $\vec{o}_2$)

$\vec{t} \leftarrow \vec{o}_1$ **and** $\vec{c}$
$\vec{f} \leftarrow \vec{o}_2$ **and** (**not** $\vec{c}$)
return ($\vec{t}$ **or** $\vec{f}$)

Lemma 3.1. *Algorithm 3.1 correctly selects the proper elements of the parameters* $\vec{o}_1$ *and* $\vec{o}_2$ *with the help of the mask* $\vec{c}$ *in constant time.*

Proof. In order to prove this masking correct it suffices to consider all possibilities. Figure 3.1 illustrates two variables a and b which hold 2 two bit wide values. The variable $mask = (true, false)$ is used to select the left value of a and the right one of b. □

4 Data Parallel Programming

This chapter shows some typical approaches in order to principally make use of SIMD instructions. The different memory layouts are demonstrated in C++.

4.1 Overview

First of all, it should be mentioned that simple arrays of *base types* do not need a special layout. It should be clear that many for-each loops which do something with each element of an array and similar operations are easy to vectorize (see also Sections 5.3 and 5.5). We concentrate in this chapter on more complex structures which typically emerge from modular programming.

As a running example a three-dimensional Vector with three *single* attributes—x, y and z—is considered. Three large equally sized arrays or other containers $vecs_1$, $vecs_2$ and $vecs_3$ with the size *size* should then be taken to perform three calculations:

1. The first one is relatively simple because the same operation needs to be done for each *single*. Hence the exact memory layout is not that important. The operation performs a *vector addition*:

   for $i = 0$ to $size - 1$ **do**
   $vecs_1[i] \leftarrow vecs_2[i] + vecs_3[i]$
   end for

2. This task is more difficult: For the z-component a subtraction must be accomplished in contrast to the x- and y-components where an addition must be performed. Hence the memory layout becomes more important. Because this calculation has more a didactic purpose than a practical use it is difficult to find a proper name. Therefore it is just referred as *mixed calculation* later on:

   for $i = 0$ to $size - 1$ **do**
   $vecs_1[i].x \leftarrow vecs_2[i].x + vecs_3[i].x$
   $vecs_1[i].y \leftarrow vecs_2[i].y + vecs_3[i].y$
   $vecs_1[i].z \leftarrow vecs_2[i].z - vecs_3[i].z$
   end for

3. The final example computation is quite complex. A *cross product* must be calculated. Different components of the input vectors must be multiplied while the

```
struct Vec3 {
    float x, y, z;
};
//...
Vec3* aos = new Vec3[size];
```

Listing 4.1: Array of Structures

intermediate results must be subtracted. These must be written to a third component of the output vector:

for $i = 0$ to $size - 1$ **do**
    $vecs_1[i] \leftarrow vecs_2[i] \times vecs_3[i]$
end for

The SSE/SSE2 instruction set introduced in Section 3.2 is exemplarily taken to discuss the different memory layouts. Once again it should be noted that aligned memory access is important. Otherwise slow unaligned moves must be used.

In order to make the descriptions as simple as possible it is assumed that the root addresses of the used data structures which hold the $vecs_1$, $vecs_2$ and $vecs_3$ are a multiple of 16. This condition can be fulfilled by using one of the techniques which is discussed in Section 5.1.

Furthermore *size* is assumed to be a multiple of four. If this boundary condition does not hold *size* must be cut to the next smaller multiple of four. Bit arithmetic as shown in Section 2.2 can be used to achieve this. Then the presented techniques can be used. The remaining calculations must be broken down to scalar calculations. The same is true if the iteration does not start at the first element (element number zero). If the starting index is not a multiple of four it must be lifted to the next multiple of four (as shown in Section 2.2). In this case the remaining elements must be computed via scalar calculations, too.

When comparing the speedup of an SIMD version to a scalar version of a calculation only the computations in one iteration are considered towards an appropriate number of iterations in the scalar version. The management of the loop is not taken into account because the exact number of instructions depends heavily on the target architecture.

4.2 Array of Structures

The most obvious approach is to simply use a *record* with three *singles* and build an array with an appropriate size (see Listing 4.1). Therefore it is called *Array of Structures* (AoS).

This must be regarded as the standard way of organizing data implied by many programming languages. The memory layout sketched in Table 4.1 emerges.

offset	0	4	8	12	16	20	24	28	32	36	40	44	···
content	x_0	y_0	z_0	x_1	y_1	z_1	x_2	y_2	z_2	x_3	y_3	z_3	···

Table 4.1: Memory layout of an AoS

The row offset indicates the offset of the corresponding content to the root address. Since the root address is a multiple of 16 each offset which is also a multiple of 16 results in an address at a 16 byte boundary. As can be seen in the table this is not necessarily the place where each new `Vec3` instance begins.

As long as simple operations have to be performed, where the same calculation must be done for all members like in the vector addition, the use of SSE instructions is possible nevertheless, because the exact memory layout can be ignored. The following pseudocode shows how this can be done:

while $size > 0$ **do**

$*vecs_1 \leftarrow *vecs_2 \oplus *vecs_3$
$*vecs_1 \leftarrow *(vecs_2 + 16 \text{ bytes}) \oplus *(vecs_3 + 16 \text{ bytes})$
$*vecs_1 \leftarrow *(vecs_2 + 32 \text{ bytes}) \oplus *(vecs_3 + 32 \text{ bytes})$

$vecs_i \leftarrow vecs_i + 48$ bytes, with $i = 1, 2, 3$
$size \leftarrow size - 4$

end while

If a more complex operation is involved like the mixed calculation an SIMD computation becomes more complicated:

while $size > 0$ **do**

$tmp_1 \leftarrow *(vecs_2 + 8 \text{ bytes}) - *(vecs_3 + 8 \text{ bytes})$
$tmp_2 \leftarrow *(vecs_2 + 20 \text{ bytes}) - *(vecs_3 + 20 \text{ bytes})$
$tmp_3 \leftarrow *(vecs_2 + 32 \text{ bytes}) - *(vecs_3 + 32 \text{ bytes})$
$tmp_4 \leftarrow *(vecs_2 + 44 \text{ bytes}) - *(vecs_3 + 44 \text{ bytes})$

$*vecs_1 \leftarrow *vecs_2 \oplus *vecs_3$
$*vecs_1 \leftarrow *(vecs_2 + 16 \text{ bytes}) \oplus *(vecs_3 + 16 \text{ bytes})$
$*vecs_1 \leftarrow *(vecs_2 + 32 \text{ bytes}) \oplus *(vecs_3 + 32 \text{ bytes})$

$*(vecs_1 + 8 \text{ bytes }) \leftarrow tmp_1$
$*(vecs_1 + 20 \text{ bytes }) \leftarrow tmp_2$
$*(vecs_1 + 32 \text{ bytes }) \leftarrow tmp_3$
$*(vecs_1 + 44 \text{ bytes }) \leftarrow tmp_4$

$vecs_i \leftarrow vecs_i + 48$ bytes, with $i = 1, 2, 3$
$size \leftarrow size - 4$

end while

```
struct Vec3 {
    float x, y, z, dummy;
};
//...
Vec3* aos = new Vec3[size];
```

Listing 4.2: Array of Structures with Dummies

The basic idea for both tasks is to add up four `Vec3` instances in one iteration. This means that in the first case only three additions are needed instead of 12 as in the scalar version. This is roughly a speedup of four. In the second computation z-components are calculated independently and written to the result locations. Thus four more operations are needed resulting in a speed up of $\frac{12}{7} \approx 1.7$. The offsets of the z-components can be comprehended in Table 4.1. The cross product cannot be written in some way in order to make reasonable use of SIMD instructions.

Although it is principally possible to speedup some calculations, this memory layout is quite impractical: One `Vec3` instance cannot be considered as one unit. This especially makes SIMD programming quite difficult, confusing and not modular.

4.3 Array of Structures with Dummies

In order to make programming more modular a dummy attribute to the *record* can be added so that size-of(`Vec3`) = 16. Listing 4.2 shows the implementation. This technique is called *AoS with Dummies*. Now one `Vec3` object always fits into one XMM register so one instance can be seen as one unit.

With an additional dummy member the memory layout looks like this:

offset	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	···
content	x_0	y_0	z_0	–	x_1	y_1	z_1	–	x_2	y_2	z_2	–	x_3	y_3	z_3	–	···

Table 4.2: Memory layout of an AoS with Dummies

In this layout every `Vec3` instance matches with a 16 byte boundary which also means that aligned moves can be used without problems. Unfortunately, 25% of the used memory is wasted this way but *size* need not to be a multiple of four which makes the use of the layout very handy in practice. Some APIs support such layouts. For example, vertices given to OpenGL can have a stride which indicates the size of each vertex. This layout can be used in the *id Tech 3* game engine, for instance.[1]

[1] See the file `./code/splines/math_vector.h` in the engine's source code. A ZIP archive of the engine can be found at `ftp://ftp.idsoftware.com/idstuff/source/quake3-1.32b-source.zip`, for

The vector addition is very simple to implement and only one instead of three additions are needed which adds up to a speedup of three:

while $size > 0$ **do**
  $*vecs_1 \leftarrow *vecs_2 \oplus *vecs_3$
  $vecs_i \leftarrow vecs_i + 16$ bytes, with $i = 1, 2, 3$
  $size \leftarrow size - 1$
end while

The second task is again more complicated and two instead of three operations are necessary which results in a speedup of 1.5:

while $size > 0$ **do**
  $tmp \leftarrow *(vecs_2 + 8 \text{ bytes}) - *(vecs_3 + 8 \text{ bytes})$
  $*vecs_1 \leftarrow *vecs_2 \oplus *vecs_3$
  $*(vecs_1 + 8 \text{ bytes}) \leftarrow tmp$
  $vecs_i \leftarrow vecs_i + 16$ bytes, with $i = 1, 2, 3$
  $size \leftarrow size - 1$
end while

Again the cross product cannot be implemented in a reasonable way in order to use SIMD instructions.

An Advantage of this technique—although maximal speedups are not achieved—is that each operation must only be implemented once. Furthermore the size of the array need not be a multiple of four in contrast to most other more advanced layouts.

4.4 Structure of Arrays

Instead of putting each `Vec3` instance sequentially into memory, each attribute can be put into a different array resulting in one array per attribute. Hence the layout is called *Structure of Arrays* (SoA). The C++ code in Listing 4.3 demonstrates this.

We now have three different arrays where the offset of the i^{th} component in each array is given by $4i$. It is assumed that the root addresses of the arrays are again a multiple of 16. Table 4.3 summarizes the memory layout.

offset	0	4	8	12	16	20	24	28	32	$\cdots$
content	x_0	x_1	x_2	x_3	x_4	x_6	x_7	x_8	x_9	$\cdots$
content	y_0	y_1	y_2	y_3	y_4	y_6	y_7	y_8	y_9	$\cdots$
content	z_0	z_1	z_2	z_3	z_4	z_6	z_7	z_8	z_9	$\cdots$

Table 4.3: Memory layout of an SoA

Alternatively, all content can be put into one large array where each stream starts at

instance; defining `FAT_VEC3` will add a fourth attribute as stated.

```
struct SoA {
    size_t size;
    float *x, *y, *z;

    SoA(size_t numElems)
        : size(numElems)
        , x( new float[size] )
        , y( new float[size] )
        , z( new float[size] )
    {}
    ~SoA() {
        delete[] x;
        delete[] y;
        delete[] z;
    }
};
```

Listing 4.3: Structure of Arrays

an offset which is dividable by 16. The `x`-, `y`- and `z`-attributes point to the beginning of each stream in the array.

Now, all three tasks are quite easy to implement. The following setup is needed for all three computations; only the loop content must be adapted:

$x_i \leftarrow vecs_i.x$, with $i = 1, 2, 3$
$y_i \leftarrow vecs_i.y$, with $i = 1, 2, 3$
$z_i \leftarrow vecs_i.z$, with $i = 1, 2, 3$
while $size > 0$ **do**
    $\langle$ *loop content* $\rangle$
    $x_i \leftarrow x_i + 16$ bytes, with $i = 1, 2, 3$
    $y_i \leftarrow y_i + 16$ bytes, with $i = 1, 2, 3$
    $z_i \leftarrow z_i + 16$ bytes, with $i = 1, 2, 3$
    $size \leftarrow size - 4$
end while

The vector addition just adds up all components in a parallel way. The loop content looks like this:

$*x_1 \leftarrow *x_2 \oplus *x_3$
$*y_1 \leftarrow *y_2 \oplus *y_3$
$*z_1 \leftarrow *z_2 \oplus *z_3$

The mixed calculation almost looks the same:

$*x_1 \leftarrow *x_2 \oplus *x_3$
$*y_1 \leftarrow *y_2 \oplus *y_3$
$*z_1 \leftarrow *z_2 \ominus *z_3$

And now the cross product can be easily implemented:

$*x_1 \leftarrow *y_2 \odot *z_3 \ \ominus \ *z_2 \odot *y_3$
$*y_1 \leftarrow *z_2 \odot *x_3 \ \ominus \ *x_2 \odot *z_3$
$*z_1 \leftarrow *x_2 \odot *y_3 \ \ominus \ *y_2 \odot *x_3$

All three tasks achieve a speedup of four and the implementation is relatively easy: Instead of scalar instructions SIMD ones are used in order to operate on *4 packed singles* in a parallel way. The setup of the loop is the same for all considered computations.

However, for large arrays the likelihood for cache misses increases and effects like paging for really large arrays can make calculations slow again because x_i, y_i and z_i which are needed during one iteration do not lie side by side in memory.

4.5 Hybrid Structure of Arrays

Another possibility is to mix the SoA with the AoS approach. Now each attribute of the *record* is a *4 packed single* instead of a *single*. This is put into an array of size *size*/4 as shown in Listing 4.4. Intel calls this layout *Hybrid SoA* [cf. Intel, a, sec. 4.5.1].

```
#define SIMD_WIDTH 4
#define SIMD_SHIFT 2 // 2^2 = 4

typedef float PackedSingles[SIMD_WIDTH];

struct HybridSoA {
    PackedSingles x, y, z;
};
// ...
HybridSoA* hSoA = new HybridSoA[ size >> SIMD_SHIFT ];
```

Listing 4.4: Hybrid SoA

Table 4.4 summarizes the resulting memory layout.

The sample computations are as easy to implement as it was for the SoA. The setup of the loop is even simpler:

while *size* > 0 **do**
    ⟨ *loop content* ⟩

offset	0	4	8	12	16	20	24	28	32	36	40	44	
content	x_0	x_1	x_2	x_3	y_0	y_1	y_2	y_3	z_0	z_1	z_2	z_3	
offset	48	52	56	60	64	68	72	76	80	84	88	92	$\cdots$
content	x_4	x_5	x_6	x_7	y_4	y_5	y_6	y_7	z_4	z_5	z_6	z_7	$\cdots$

Table 4.4: Memory layout of a Hybrid SoA

$vecs_i \leftarrow vecs_i + 48$ bytes, with $i = 1, 2, 3$
$size \leftarrow size - 4$
end while

The vector addition can be implemented like this:

$vecs_1{\rightarrow}x \leftarrow vecs_2{\rightarrow}x \oplus vecs_3{\rightarrow}x$
$vecs_1{\rightarrow}y \leftarrow vecs_2{\rightarrow}y \oplus vecs_3{\rightarrow}y$
$vecs_1{\rightarrow}z \leftarrow vecs_2{\rightarrow}z \oplus vecs_3{\rightarrow}z$

The mixed calculation task is as easy:

$vecs_1{\rightarrow}x \leftarrow vecs_2{\rightarrow}x \oplus vecs_3{\rightarrow}x$
$vecs_1{\rightarrow}y \leftarrow vecs_2{\rightarrow}y \oplus vecs_3{\rightarrow}y$
$vecs_1{\rightarrow}z \leftarrow vecs_2{\rightarrow}z \ominus vecs_3{\rightarrow}z$

And for the cross product nothing special must be considered, too:

$vecs_1{\rightarrow}x \leftarrow vecs_2{\rightarrow}y \odot vecs_3{\rightarrow}z \ \ominus \ vecs_2{\rightarrow}z \odot vecs_3{\rightarrow}y$
$vecs_1{\rightarrow}y \leftarrow vecs_2{\rightarrow}z \odot vecs_3{\rightarrow}x \ \ominus \ vecs_2{\rightarrow}x \odot vecs_3{\rightarrow}z$
$vecs_1{\rightarrow}z \leftarrow vecs_2{\rightarrow}x \odot vecs_3{\rightarrow}y \ \ominus \ vecs_2{\rightarrow}y \odot vecs_3{\rightarrow}x$

Like in the SoA technique all three calculations gain a speedup of four but this time all needed data for one SIMD computation lie side by side in memory which decreases the likelihood for cache misses and the need for paging. Furthermore the loop overhead shrinks compared to the loop necessary for the SoA approach.

4.6 Converting Back and Forth

Once a decision is made which data structure is used it is best to stick with this one as converting from an AoS to a SoA, for instance, implies an overhead of linear running time. Furthermore, a conversion using the same chunk of memory can be tricky in many programming languages. The use of a different copy is easier to implement but consumes a linear amount of memory which also increases the likelihood of cache misses or paging for really large arrays.

Another option is to convert the data on the fly. But this implies an overhead of linear running time for each loop. An on-the-fly-conversion may or may not be faster in the end.

4.7 Summary

	Vector Addition	Mixed Calculation	Cross Product
AoS	4	1.7	n/a
AoS with Dummies	3	1.5	n/a
SoA	4	4	4
Hybrid SoA	4	4	4

Table 4.5: Speedup of the three sample computations with different memory layouts

Table 4.5 shows the theoretically possible speedups of the three sample tasks with the different discussed memory layouts. The following list summarizes the different advantages and disadvantages of them:

AoS: This is the standard layout used in most APIs. Although SIMD computations are possible for some calculations the implementation can be quite tricky.

AoS with Dummies: Although this layout is even slower than the AoS version and wastes 25% of memory it is in practice currently perhaps the most convenient way to integrate SIMD instructions since the implementation is much easier compared to all other layouts.

SoA: This layout is a simple possibility to make full use of SIMD instructions. However, cache misses or even paging for large arrays become likely. Additionally, the loop setup is a bit expensive as every attribute is an independent array and thus a pointer to each attribute must be adjusted in each iteration of the loop.

Hybrid SoA: The setup of this layout is slightly more difficult than for a SoA but data needed in one computation is held in one chunk of memory. Cache misses and paging occur in this constellation less frequently than with an SoA because the data locality is better in this layout. The loop setup is less expensive than in the SoA approach because all data are located in one array.

The Hybrid SoA seems to be the best layout. Unfortunately, much work is needed with current programming languages to support this technique.

5 Current SIMD Programming Techniques

While the last chapter discussed different memory layouts suitable for SIMD computations in a more abstract way, this chapter focuses on different techniques available for programmers in order to make use of SIMD instructions. Four criteria are used to discuss the different techniques:

Modularity: A routine should only be programmed once. The need for a special SIMD version of a routine increases the code base and hence the amount of work and the likelihood of bugs. Furthermore the program is often more difficult to understand.

High-Level Programming: The programmer should deal with as less low-level details as possible. Low-level programming techniques are often more verbose and more difficult to understand which again increases the amount of work and the likelihood of bugs.

Portability: It is also desirable that—whatever the programmer does to make use of SIMD technologies—it works on miscellaneous SIMD machines.

Performance: All work is not worth the trouble if the result is not faster than the scalar version. Thus a significant performance gain must be expected.

5.1 Getting Aligned Memory

A common problem in most approaches is to get aligned memory in order to use faster aligned moves.

5.1.1 Aligned Stack Variables

The GNU Compiler Collection (GCC) allows this construct:

```
typedef float __attribute__ ((aligned (16))) PackedSingles;
```

Most other C and C++ compilers provide similar enhancements. C++0x will add a standardized support for this request. Internally, this means that the stack must be aligned on a 16 byte boundary while the size of all local variables must be a multiple of 16 or an appropriate amount of padding bytes must be inserted. Of course, variables with more relaxed boundary conditions can also be used as padding space. For x64 the stack can be aligned with

```
void* aligned_malloc(size_t size, size_t align) {
    void *pa;
    void *p = malloc(size + align-1 + sizeof(void*));
    if (!p) return 0;

    pa = (void*) (((size_t) p + sizeof(void*) + align-1)
            & ~(align-1));
    *(((size_t*) pa)-1) = (size_t) p;

    return pa;
}

void aligned_free(void* p) {
    free((void*) *(((size_t*) p)-1));
}
```

Listing 5.1: Aligned malloc and free

andq \$0xFFFFFFFFFFFFFFF0, %**rsp**

since the stack grows decreasingly. Once the stack is aligned all functions which leave the stack in an aligned state, too, can be used without explicitly aligning the stack again.

5.1.2 Aligned Heap Variables

Unfortunately, above mentioned syntax only guarantees the proper alignment of variables allocated on the stack. When requesting dynamic memory on the heap other OS-/vendor specific functions like `posix_memalign`[1] or `_aligned_malloc`[2] must be used.[3] If such functions are not available or maximum portability should be achieved other techniques must be used. The C code in Listing 5.1 shows how an `aligned_malloc` and an `aligned_free` can be implemented on top of `malloc` and `free` found in the C standard library.

First enough memory is fetched at address `p` with a standard `malloc`. In the worst case this is the alignment minus one. As the original pointer should be stored in this chunk, too, **sizeof**(**void***) more bytes needs to be fetched. The original address should be stored directly **sizeof**(**void***) bytes below the actual aligned address. Hence the proper aligned address is found with $p + ptr\_size \bmod alignment$ or with special bit arithmetic when

[1] See http://www.opengroup.org/onlinepubs/000095399/functions/posix_memalign.html.
[2] See http://msdn.microsoft.com/en-us/library/8z34s9c6(VS.80).aspx.
[3] For more information on the reasons for this have a look at http://gcc.gnu.org/bugzilla/show_bug.cgi?id=15795.

alignment is assumed to be a power of two as shown is Section 2.2. At last the original address gets stored at the proposed position. The `aligned_free` procedure just needs to get this address and to invoke a standard `free` with it.

Note that in C++ such issues can be hidden by overloading the **new**, **new**[], **delete** and **delete**[] operators. Using specialized allocators may also help.

5.2 Assembly Language Level

The best control over the generated machine code can be achieved by programming directly in assembly language. The disadvantages when dealing with assembly code are commonly known and are not repeated here. Several techniques are available to the programmer—each one with its own advantages and culprits.

5.2.1 The Stand-Alone Assembler

The stand-alone assembler is the classic assemblers which assembles the source code to machine code/object code. Modern assemblers have powerful macro features and even small programming languages can be build on top of some macro facilities. Not the whole program must be written in assembly language in order to take advantage of these tools. Just some subroutines can be implemented directly in assembly language. Special care must be taken when designing the interface between the high-level language and the assembly language module. Otherwise crashes or linker errors can occur.

Function calls can cause a significant overhead which can be easily overseen. A highly optimized assembly language subroutine can actually cause a slower running time than the high-level pendant where the compiler is able to inline the function and eliminate the function call overhead. However, subroutines with a large running time can disregard this effect.

5.2.2 The Inline-Assembler

In this scenario the programmer just writes small portions of the program in assembly language which is directly embedded into the high-level language. Unfortunately, the exact syntax of the inline-assembler is not standardized. So inline assembly language code snippets are usually not even portable between different compilers. Besides that the compiler does not know how registers are used in the inline assembly snippet. Therefore it is necessary to save all used registers beforehand and restore the old values afterwards.

GCC circumvents the problem with the extended inline assembly syntax. The programmer can explicitly say which class of registers he wants to use for his variables, which variables he wants to read from memory and which registers get implicitly clobbered. Furthermore, constraints to each in and out value must be given, so GCC knows

```
typedef float v4ps __attribute__ ((__vector_size__ (16)));

void add(size_t size, float* dst, float* src1, float* src2) {
    v4ps tmp;
    if (!size) return;

    asm("shl %3, $2    \n\t" /* size /= 4     */
        "loop:         \n\t"
        "movaps %0, %4\n\t" /* *src1 -> tmp */
        "addps  %1, %4\n\t" /* tmp += *src2 */
        "movaps %4, %3\n\t" /* tmp -> dst    */
        "addl $16, %0 \n\t" /* src1 += 16    */
        "addl $16, %1 \n\t" /* src2 += 16    */
        "addl $16, %2 \n\t" /* dst  += 16    */
        "decl %3       \n\t" /* --size        */
        "jnz loop      \n\t" /* goto loop if size != 0 */
        : "=m" (src1), "=m" (src2), "=m" (dst), "=r" (size)
        : "x" (tmp) /* get temporary xmm register */
        : /* no clobbers */
        );
}
```

Listing 5.2: Extended inline assembly syntax

which register copies are necessary. E. Kunst and J. Quade [2009], e.g., give a detailed introduction to GCC's extended inline assembly language.

Listing 5.2 shows how to make use of the **addps** instruction with the extended inline assembly syntax. The constraints must be chosen very carefully otherwise it may occur that the code works when building without optimization but fails to work when building with optimization turned on or even different GCC versions may result in correct or incorrect code.

Above all the extended inline assembly syntax must be regarded as very cryptic. Even normal assembly language pieces are usually difficult to understand. With the extended inline assembly syntax this issue gets even worse. The special letters and symbols, which are used as constraints, register classes etc., are far from being self-explanatory. Many combinations and corner cases of possible chosen registers must be simultaneously considered when reading the code. On the other hand well written inline assembly code smoothly integrates with the compiler and does not interfere with its optimization passes.

```
#include <xmmintrin.h>

void add(size_t size, __m128* dst,
         __m128* src1, __m128* src2) {
    for (size >>= 2; size; --size, ++dst, ++src1, ++src2)
        *dst = _mm_add_ps(*src1, *src2);
}
```

Listing 5.3: Use of a compiler intrinsic

5.2.3 Compiler Intrinsics

Another approach is to use so-called *compiler intrinsics*. These are small wrappers upon machine instructions. Although technically this is not directly assembly language the programmer has to deal with machine instruction nevertheless. Listing 5.3 shows how to make use of the **addps** instruction. This basically results in almost the same assembly code than the previous technique rendering the use of compiler intrinsics in most scenarios as the recommend approach since the code is much more readable, maintainable and many difficulties like register allocation are handled by the compiler.

Carefully written programs which make use of SSE compiler intrinsics are even portable between x86 an x64. Generally, compiler intrinsics must not be regarded as portable, of course. Different compiler vendors or even different versions of the same compiler can cause problems.

5.2.4 Summary

As already said the culprits of dealing directly with assembly language are well known. Unfortunately, in many scenarios this is the only way to make use of SIMD instructions. If the programmer has to use the assembly language level compiler intrinsics are recommend.

As the programmer has direct control of the used machine instructions maximum performance can be achieved. However, the programmer has to deal with many low-level details and special SIMD routines must be written. Thus the code is not modular. It may happen that the code is not even portable between different CPUs of the same architecture family, when special instruction sets are used, which are not available for all members of the family.

```
void add(size_t size, float* restrict dst,
         float* restrict src1, float* restrict src2) {
    for (size >>= 2; size; --size, ++dst, ++src1, ++src2)
        *dst = *src1 + *src2;
}
```

Listing 5.4: Use of the auto vectorizer

5.3 The Auto-Vectorizer

Some modern compilers like GCC and the Intel C Compiler (ICC) have an *auto-vectorizer* which has already been briefly mentioned in Chapter 1. This optimization pass tries to transform a loop with scalar calculations to a loop with SIMD instructions. For a compiler it is very difficult to resolve data dependencies and to proof a pointer to be properly aligned. Possible aliasing pointers may prohibit the compiler from some optimizations. The **restrict** keyword from the C99 standard may help in this case. ICC and GCC have special compiler switches which help the programmer to find out whether a loop was vectorized. For this reason it can be quite challenging to write a loop in a way so that the compiler can transform it in a feasible way as shown in Listing 5.4.

Although GCC's auto-vectorizer officially supports aligned access[4] the author was not able to use this feature. Instead an error message was emitted:[5]

```
error: alignment of array elements is greater than element size
```

Even if the compiler is able to transform the loop, the resulting code does not compare well with other techniques presented in this chapter while in some cases a nice speedup can be achieved. On the other hand the programmer does not need to deal with low-level details and the source code is portable.

5.4 SIMD Types

OpenCL, GLSL, Cg, HLSL and other shader languages have built-in types for SIMD calculations. These types are called *vector* or *SIMD types*. Many C and C++ compilers provide SIMD types as extensions. Usually element-wise operations are available as demonstrated in Listing 5.5. The resulting assembly code makes use of aligned moves and the **addps** instruction.

A problem is that the programmer must know the vector length of the target machine. Changing the vector size from 16 to 8 in Listing 5.5, for instance, has the effect that

[4] See http://gcc.gnu.org/projects/tree-ssa/vectorization.html for more information of the supported features.

[5] Example 3 from the previously mentioned homepage was tried with GCC versions 4.1.2 and 4.4.3.

```
typedef float v4ps __attribute__ ((__vector_size__ (16)));

void add(size_t size, v4ps* dst, v4ps* src1, v4ps* src2) {
    for (size >>= 2; size; --size, ++dst, ++src1, ++src2)
        *dst = *src1 + *src2;
}
```

Listing 5.5: Use of SIMD types

GCC breaks down the operation to scalar computations.[6] Thus the programmer has to deal with low-level details and the metaphor of SIMD types only provides a poor abstraction level. The performance also depends on the target machine and the chosen vector types in the source program. Poorly chosen vector sizes or targeting machines with different internal vector sizes can reduce the performance. On the other hand if the types are wisely chosen the performance is great. Principally, the code is portable but a performance penalty may occur. Modular programming is not possible as again vectorized operations and scalar ones must be implemented.

5.5 Vector Programming

Languages like APL, Vector Pascal and Matlab/Octave support specialized calculations with arrays. For this reason such languages are called *array*, *multidimensional* or *vector programming languages*.

Let s_r, s_1, $s_2 \in T_{base}$ and a_r, a_1 and a_2 be *arrays of* Ts. Now the basic idea is that a function like $f : T_{base} \times T_{base} \to T_{base}$ can be used in four different ways:

- All arguments are scalar values and a scalar value is returned:

$$s_r \leftarrow f(s_1, s_2)$$

- All arguments are equally sized arrays. The function f is element-wisely applied while a result array of the same size is created:

$$a_r \leftarrow f(a_1, a_2)$$

- The first argument is an array whereas the second one is a scalar value. In this case f is applied on each element of a_1 while each time s_1 is used as input and a result array a_r is formed:

$$a_r \leftarrow f(a_1, s_1)$$

[6] This has been tested with GCC 4.4.3 on x64.

```
a = rand(1, 1000);
b = rand(1, 1000);
res = a + b;
res += 2.0;
```

Listing 5.6: Octave program which makes use of vector pgramming

- This is the same case as above but this time the first argument is the array and the second one the scalar one:

$$a_r \leftarrow f(s_1, a_1)$$

Listing 5.6 shows just a few things which are possible in Matlab/Octave: First of all, two arrays `a` and `b` with 1000 elements each are created and initialized with random *floats* within the interval $[0, 1]$. Then an equally sized result array `res` is created by an element-wise addition of `a` and `b`. Afterwards each element of `res` is increased by 2.0. It is obvious that compilers can easily generate SIMD instructions from such programs. Especially, Vector Pascal has a rich set of features which work well with SIMD instructions.

More sophisticated techniques of vector programming include multidimensional arrays, specifying consecutive elements of an array as sub-array, array slicing (extracting elements with different indices or dimensions) and reduce operations, which reduce the number of dimensions of the input data by one or more dimensions.

Libraries of other languages try to emulate these features with a smart set of library functions. For example, in C++ the container `std::valarray` of the C++ standard library is basically an array with many overloaded operators [see Stroustrup, 2000, sec. 22.4]. It can be used to code as if using a language which directly supports vector programing. However, `valarray` implementations may or may not have template specializations optimized for SIMD instructions available. The *macstl* project[7] for PowerPC, for instance, provides specialized `valarray` versions which make use of the AltiVec instruction set. Most implementations and C++ compilers, however, are far away from the quality of what can be achieved with specialized vector programing languages. The problem is that trying to implement SIMD support on top of a language is a very tricky task which bears many subtle problems: A feature-rich language and a mix of assembly language, compiler intrinsics (see Section 5.2.3), SIMD types (see Section 5.4) and a very good compiler with smart optimization passes is needed.

Vector programming turns out to be a portable, efficient and high-level way of programming in order to make use of SIMD instructions. Unfortunately, this only works for arrays of *base types*. When *records* are involved there is no modular way to handle them. The programmer is still forced to write scalar versions of functions which operate

[7] See http://www.pixelglow.com/macstl/.

on records and must find an appropriate data structure as discussed in Chapter 4 with a set of special SIMD functions which operate on these streams.

5.6 Use of Libraries

If optimized libraries which make use of SIMD instructions are available for a given problem, programming becomes easy, of course. Besides `std::valarray` which has already been discussed in the last section, the Eigen library[8] and several BLAS (Basic Linear Algebra Subprograms) implementations[9] are good examples. However, as libraries must also be implemented in some way this is not a valid option from a theoretical perspective. Furthermore, the license of a particular library may not be suitable for the needs of the programmer. As development of several SIMD related projects ceased[10] the design of such a library must be regarded as a quite complex task as already outlined in the last section.

5.7 Summary

	Modular	High-Level	Portable	Performance
Stand-Alone Assembler	–	–	–	✓
Inline-Assembler	–	–	–	✓
Compiler Intrinsics	–	–	–	✓
Auto-Vectorizer	–	○	✓	○
SIMD Types	–	–	✓	○/✓
Vector Programming	–	✓	✓	✓

Table 5.1: Summary of different SIMD programming techniques

Table 5.1 summarizes the discussed approaches and tries to assign a judgement of each criteria. The following list gives a short justification and some comments:

Stand-Alone Assembler: Subroutines directly coded in assembly language can be very fast. It is obvious that this code is neither modular nor portable. Furthermore, the programmer must deal with many subtle low-level details.

[8] See http://eigen.tuxfamily.org.
[9] See http://www.netlib.org/blas/.
[10] Examples are libSIMDx86 (see http://simdx86.sourceforge.net/) or libSIMD (see http://libsimd.sourceforge.net/), for instance. Development of the earlier mentioned macstl also stopped.

Inline Assembler: Everything which has just been said about the stand-alone assembler is also true for the inline assembler. GCC's extended inline assembler syntax allows even small inline assembler snippets which do not hurt the performance.

Compiler Intrinsics: Often, the use of compiler intrinsics is as fast as directly using assembly language while some low-level details are ceded to the compiler. But programming with intrinsics can still be very tricky.

Auto-Vectorizer: While it is principally possible to generate fast code with this compiler optimization pass, this is in practice rarely the case. The compiler has difficulties to use aligned moves and to resolve data dependences.

SIMD Types: Abstract support of SIMD types makes the code portable. However, inappropriate chosen types can reduce the performance. Properly chosen types can result in very fast code which is comparable to hand written assembly language code so this is a bit hard to judge with a simple table. Modular high-level programming is not possible with this technique, neither.

Vector Programming: This is a portable and high-level programming technique with good performance. Unfortunately, everything beyond simple streams of scalar data must be handled specially.

All techniques suffer from the fact, that special code must be written for data structures beyond simple arrays of *floats* or *integers*. This makes modular programming impossible. Besides that vector programming is the most convenient tool in order to do SIMD programming from a programmer's perspective. Furthermore, it would be even better, if the Hybrid SoA could be used automatically.

6 Language Supported SIMD Programming

As discussed in Chapter 4 the best memory layout for SIMD computations is a Hybrid SoA, while the most convenient way of doing SIMD programming is the use of vector languages. The approach, which is presented in this chapter, tries to combine the memory layout with vector programming in order to extend this technique, so that it also works for *records*.

6.1 Overview

The basic idea is to use Hybrid SoA instead of ordinary arrays. As we must distinguish between the whole data structure and one instance of a *record* with packed types we call the whole data structure *SIMD container*. The vector version of the *record* which is used internally is called *vectorized record*. The programmer can simply choose to use an SIMD container instead of an array. Now, some functions which should also be usable with vectorized records instead of ordinary ones, must be vectorized by the compiler. Hence a second version of such a function is needed. The programmer must indicate this wish by an appropriate keyword, for instance. At last, the data structure and the vectorized function must be put together in *SIMD statements* which are basically loops over SIMD containers while vectorized functions instead of scalar ones are used.

The Swift program in Listing 6.1 which implements the three sample computations of Section 4.1 shows how this paradigm can look in a high level language:

The **simd** keyword in front of **class** in line 1 indicates that the given data structure `Vec3` should be usable within SIMD containers. Therefore the compiler first checks whether this is possible and generates a second vectorized record. The **simd** keyword in front of **operator** and **routine** in lines 8, 14 and 20 have a different meaning. In this pace the compiler checks whether vectorization of these routines is possible and generates a second version which takes vectorized `Vec3` instances instead of plain `Vec3` instances as parameters and result values. Lines 28–30 create three SIMD containers of `Vec3` with size `size`. Finally, the SIMD statements in lines 32–34 make use of the vectorized data structure and routines. For all elements in the SIMD containers the three sample computations are performed.

```
1  simd class Vec3
2      real x
3      real y
4      real z
5
6      #...
7
8      simd operator + (Vec3 v1, Vec3 v2) -> Vec3 result
9          result.x = v1.x + v2.x
10         result.y = v1.y + v2.y
11         result.z = v1.z + v2.z
12     end
13
14     simd routine mixed(Vec3 v1, Vec3 v2) -> Vec3 result
15         result.x = v1.x + v2.x
16         result.y = v1.y + v2.y
17         result.z = v1.z - v2.z
18     end
19
20     simd routine cross(Vec3 v1, Vec3 v2) -> Vec3 result
21         result.x = v1.y * v2.z - v1.z * v2.y
22         result.y = v1.z * v2.x - v1.x * v2.z
23         result.z = v1.x * v2.y - v1.y * v2.x
24     end
25
26     routine foo()
27         # ...
28         simd{Vec3} vecs1 = size
29         simd{Vec3} vecs2 = size
30         simd{Vec3} vecs3 = size
31         # ...
32         simd[0x, size]: vecs1 = vecs2 + vecs3
33         simd[0x, size]: vecs1 = Vec3::cross(vecs2, vecs3)
34         simd[0x, size]: vecs1 = Vec3::mixed(vecs2, vecs3)
35     end
36 end
```

Listing 6.1: Language extensions for modular SIMD support

6.1.1 SIMD Width

The following vectorization algorithms need to know the register width of the target SIMD architecture. This is assumed to be a global constant power of two called $simd_{width}$. In the case of SSE this would be 16 since the width of an XMM register is 16 bytes. In the case of 3D-Now! this would be eight as the used MMX registers have an width of eight bytes. If several options are available the best one should be chosen. This is usually the one with larger registers. In the case of 3D-Now! and SSE the latter one would be chosen while 3D-Now! is not used at all. Usually a better instruction set supersedes the less powerful one rendering the old technology as deprecated. For instance, AMD officially does not recommend the use of 3D-Now! instructions in favor of 128-Bit Media Instructions [see AMD, a, pg. 4] anymore. Furthermore, we assume that $simd_{width}$ is greater than all *base types* so resulting *packed types* have at least two elements.

6.1.2 Summary

As already outlined the vectorization works on three stages. These are

1. vectorization of the necessary data structures,
2. vectorization of the caller code and
3. vectorization of the callee code.

The following sections show in detail how this can be implemented.

6.2 Vectorization of Data Structures

Since not all types are vectorizable the programmer must give a hint to the compiler when declaring own types i.e., *records*.

6.2.1 SIMD Length of Types

If the programmer declares a *record* and asks the compiler, that it should be possible to use this *record* within an SIMD container, a special keyword, which indicates this wish, must be used. Since the vectorization of a *record* is not necessarily possible the compiler checks if this wish can be fulfilled and emits an error message otherwise. Algorithm 6.1 calculates a length of a type. This is used to decide whether a type is vectorizable:

Definition 6.1 (Vectorizable Type)**.** *A* type $t \in T_{all}$ *is called* vectorizable *if holds:*

$$\textsc{LengthOf}(t) \geq 0 \ .$$

This means that all types involved in a data structure must have the same size. An exception is the *boolean* type: It just leaves the choice to the other involved types if available or stays at 0 if just *booleans* are involved. The special value -1 means that a type is not vectorizable.

Lemma 6.1. *Let t be a vectorizable type and $simd_{length} > 0$ a constant power of two. Now it holds*

1. $\textsc{LengthOf}(t) = simd_{length} \vee \textsc{LengthOf}(t) = 0$ *if* $t \in T_{base}$ *or*
2. *the first case is true for each attribute (id, t_a) of t.*

The algorithm runs in $\mathcal{O}(n)$ time where n is the number of all recursively enumerated attributes of t if $t \in T_{record}$ or in $\mathcal{O}(1)$ time otherwise. This running time is optimal.

Remark. The second case implies that $t \in T_{record}$. Otherwise t would not have been vectorizable in the first place.

Proof. Algorithm 6.1 assures that case 1 is true for all $t \in T_{base}$. We prove by structural induction that case 2 is also true:

The proposition is true for all $t \in T_{record}$ which does not contain any subrecords as the loop ensures that either all attributes are *boolean* typed which means an SIMD length of 0 or at least one attribute is not a *boolean*. In this case all *integer* or *float* typed attributes have the same size s. An SIMD length of $simd_{width}/s$ derives. All other cases imply that t is not vectorizable which violates the precondition.

Assume that the proposition holds for all subrecords of t. Now the loop ensures that all *integer*, *float*, *boolean* or *record* typed attributes have the same SIMD length $simd_{length}$

Algorithm 6.1 LengthOf(t)

```
if t ∈ T_int ∪ T_float then
    return simd_width ÷ size-of(t)
else if t = boolean then
    return 0
else if t ∈ T_record then
    simd_length ← 0
    for all attributes (id, t_a) of t do
        tmp = LengthOf(t_a)
        if tmp = −1 then
            return −1
        else if simd_length = 0 then
            simd_length ← tmp
        else if tmp ≠ 0 ∧ tmp ≠ simd_length then
            return −1
        end if
    end for
    return simd_length
else
    return −1
end if
```

or have the SIMD length 0. All other cases again imply that t is not vectorizable. Thus the proposition is also true for such a t.

In the case of $t \in T_{all} \backslash T_{record}$ the result is directly computed in constant time which is obviously optimal. With $t \in T_{record}$ Algorithm 6.1 recursively loops over all attributes of t while the rest of loop body runs in constant time. This means a running time of $\mathcal{O}(n)$. Since every attribute must be examined this running time is optimal. □

6.2.2 Vectorization of Types

Now, after a type has been proven to be vectorizable, its vectorized type along with its SIMD length $simd_{length}$ can be determined as shown in Algorithm 6.2:

Each member of a record is recursively examined until a *base type* t has been found. Then a new appropriate *packed type*—an $simd_{length}$ *packed* t to be precise—is used in the vectorized *record*.

Definition 6.2 (Vectorized Type and Its SIMD Length)**.** *Let t be a vectorizable type and*

$$(simd_{length}, t_{vec}) \coloneqq \text{VecType}(t, \text{LengthOf}(t)) \ .$$

Algorithm 6.2 VecType(t, n)

if $n = 0$ **then**
    $simd_{length} \leftarrow simd_{width}$
else
    $simd_{length} \leftarrow n$
end if
if $t \in T_{int} \cup T_{float}$ **then**
    return ($simd_{length}$ *packed* t, $simd_{length}$)
else if $t =$ *boolean* **then**
    $s \leftarrow simd_{width} \div simd_{length}$
    $t_{uint} \leftarrow$ appropriate *unsigned integer* with size-of(t_{uint}) $= s$
    return ($simd_{length}$ *packed* t_{uint}, $simd_{length}$)
else ▷ $t \in T_{record}$
    create new empty record r_{vec}
    for all attributes (id, t_a) of t in descending order **do**
        ($t_{a_{vec}}$, *foo*) $\leftarrow$ VecType(t, $simd_{length}$) ▷ *foo* $\equiv simd_{length}$
        append attribute (id, $t_{a_{vec}}$) to r_{vec}
    end for
    return (r_{vec}, $simd_{length}$)
end if

We say t_{vec} is t's vectorized type with SIMD length $simd_{length}$.

Lemma 6.2. *Let t_{vec} be t's vectorized type with its SIMD length $simd_{length}$ and $s > 0$ be a constant power of two. Now $simd_{length}$ is also a constant power of tow and it holds:*

$$simd_{length} = \begin{cases} simd_{width}, & \text{if } \textsc{LengthOf}(t) = 0 \\ \textsc{LengthOf}(t), & \text{otherwise.} \end{cases}$$

Furthermore for t_{vec} holds:

1. num-elems-of$(t_{vec}) = simd_{length}$ *and* size-of$(t_{vec}) = s$ *if* $t \in T_{base}$ *or*
2. *the first case is true for each attribute* (id, t_a) *of* t_{vec}.

The algorithm runs in $\mathcal{O}(n)$ time where n is the number of all recursively enumerated attributes of t if $t \in T_{record}$ or in $\mathcal{O}(1)$ time otherwise. This running time is optimal.

Proof. The proof is broken down into several parts:

- Analog to the proof of Lemma 6.1 the proposed running time is shown to be correct and optimal.
- The proposition for $simd_{length}$ is true as the very first if-else case in Algorithm 6.2 guarantees this condition.
- Trivially, the Algorithm 6.2 is correct for all $t \in T_{base}$.
- Since t must be vectorizable by the precondition, for each recursively enumerated attribute (id, t_a) of t holds either $\textsc{LengthOf}(t_a) = simd_{length}$ or $\textsc{LengthOf}(t_a) = 0$ by Lemma 6.1 if $t \in T_{record}$.

We now must distinguish two cases:

$\textsc{LengthOf}(t_{vec}) > 0$: In this case each recursively enumerated attribute is vectorized to a packed type with $simd_{length} = simd_{width}/\textsc{LengthOf}(t_{vec})$ many elements. Since $simd_{width}$ as well as $\textsc{LengthOf}(t_{vec})$ are both powers of two and only types which sizes are a power of two are considered, both $simd_{length}$ and the size of the vectorized type are also powers of two.

$\textsc{LengthOf}(t_{vec}) = 0$: In this corner case each recursively enumerated attribute is vectorized to a packed type with $simd_{length} = simd_{width}$ many elements. Similar to the previous case, both $simd_{length}$ and the size of the resulting vectorized type are powers of two.

□

Currently, a *record* which solely consists of *booleans* gets an SIMD length of $simd_{width}$ assigned. As an alternative it may be reasonable to let the programmer decide which SIMD length such a *record* should have. A special language construct could be implemented which indicates this wish.

From a vectorized type t_{vec} we can get its source type just as well. We call this algorithm DeVecType(t_{vec}) where t_{vec} is a vectorized type and the algorithm returns t_{vec}'s source type. We also call t a *scalar type* and t_{vec} a *vector type*.

After a preprocessing all k *records* which need vectorization can be vectorized and the source types can store a link to its corresponding vectorized type and vice versa. Hence after this preprocessing algorithm VecType and DeVecType can both run in constant time. Since all *records* must be examined anyway in a compiler this overhead does not increase the overall asymptotic running time. We express this by the following lemma:

Lemma 6.3. *After a preprocessing of $\mathcal{O}(kn)$ time with k* records *which need vectorization,* VecType *and* DeVecType *can be implemented with constant running time in all cases.*

6.2.3 Extract, Pack and Broadcast

Later on we need a few operations between a value v_{vec} of a vector type t_{vec} and a value v of a scalar type t along with an index $0 \leq i < simd_{length}$ where $(t_{vec}, simd_{length}) =$ VecType(t).

Extract(i, v_{vec}, t_{vec}): Algorithm 6.3 returns v where the index i is used to recursively get the i^{th} element of the involved values with *packed types*.

Pack(i, v, v_{vec}, t_{vec}): Similarly, Algorithm 6.4 sets v recursively into the i^{th} position of v_{vec}. The updated value is returned.

Broadcast(v, v_{vec}, t_{vec}): If recursively all elements of v_{vec} must be set to v Algorithm 6.5 can be used.

So-called *shuffle* or *swizzle instructions* are usually available on SIMD capable processors which can be used to implement these algorithms. Depending on the ISA of the target machine some optimizations may be available—especially for the Broadcast operation.

6.2.4 SIMD Containers

We can now exactly define SIMD containers:

Definition 6.3 (SIMD Container)**.** *Let t be a vectorizable type and*

$$(simd_{length}, t_{vec}) \coloneqq \text{VecType}(t, \text{LengthOf}(t)) \ .$$

Algorithm 6.3 EXTRACT(i, v_{vec}, t_{vec})

$t \leftarrow$ DEVECTYPE(t_{vec})
if $t_{vec} \in T_{packed}$ **then**
    return $v_{vec}[i]$
else ▷ $t_{vec} \in T_{record}$
    create value v of type t
    for all attributes $a = (id, t_{vec_a})$ of t_{vec} **do**
        let v_{vec_a} be v_{vec}'s associated subvalue
        set v's corresponding subvalue to EXTRACT(i, v_{vec_a}, t_{vec_a})
    end for
    return v
end if

Algorithm 6.4 PACK(i, v, v_{vec}, t_{vec})

$t \leftarrow$ DEVECTYPE(t_{vec})
$v_{vec_u} \leftarrow v_{vec}$
if $t_{vec} \in T_{packed}$ **then**
    $v_{vec_u}[i] \leftarrow v$ ▷ if $t = boolean$ $v_{vec_u}[i]$ must be set to $00\ldots00_{16}$ or $\mathrm{FF}\ldots\mathrm{FF}_{16}$
    return v_{vec_u}
else ▷ $t_{vec} \in T_{record}$
    for all attributes $a = (id, t_{vec_a})$ of t_{vec} **do**
        let v_{vec_a} be v_{vec}'s associated subvalue
        let v_a be v's associated subvalue
        set v_{vec_u}'s corresponding subvalue to PACK(i, v_a, v_{vec_a}, t_{vec_a})
    end for
    return v_{vec_u}
end if

Algorithm 6.5 BROADCAST(v, v_{vec}, t_{vec})

$v_{vec_u} \leftarrow v_{vec}$
for all $i = 0$ to $simd_{length} - 1$ **do**
    $v_{vec_u} \leftarrow$ PACK(i, v, v_{vec_u}, t_{vec})
end for
return v_{vec_u}

An SIMD container of type t with n elements consists of $\lceil \frac{n}{simd_{length}} \rceil$ many t_{vec} instances which lie side by side in memory. A record with two attributes is used to keep account of this data structure:

$$\langle (ptr,\ pointer),\ (size,\ t_{size}) \rangle\ .$$

The attribute ptr points to the beginning of the t_{vec} instances while size is set to n. A unit of $simd_{length}$ many elements, while the index of the first element is a multiple of $simd_{length}$, is called SIMD block.

The vector loop which is discussed in the next section fetches an entire SIMD block in a single iteration. If single values must be accessed the PACK and EXTRACT operations presented in the last section can be used. Additionally, standard techniques which are also used to dynamically increase or shrink an array apply to this data structure just as well [see Cormen et al., 2001, sec. 17.4].

6.3 Vectorization of the Caller Code

An SIMD statement consists of a prefix which indicates the bounds of a loop and a statement which is the actual calculation. A loop which loops over $[j, k[$ may look like this:

```
simd[j, k]: statement
```

Of course, other notations are possible. Especially, the use of [j, k] to indicate an interval of $[j, k[$ may look irritating at first glance but such loops are very common in practice and a notation of [j, k[may cause problems with nesting.

Not every SIMD statement is vectorizable and the compiler should throw an error message if a nonvectorizable SIMD statement is used. In programming languages a statement usually consists of a root expression. We use this to define SIMD expressions and vectorizable statements:

Definition 6.4 (SIMD Expresssion)**.** *An* SIMD expression *is*

- *an expression returning an SIMD container,*
- *an SIMD index,*
- *an expression which is specially marked as an SIMD expression or*
- *an expression which has at least one SIMD expression as child.*

All other expressions are called scalar expressions.

Remark. The programmer needs a way to force the use of SIMD operations instead of scalar ones. This can be done via a special keyword like **simd**(expr), for instance. The exact semantics of an SIMD index are defined in Section 6.3.2.

Definition 6.5 (SIMD Statement)**.** *A statement is called* vectorizable *with* SIMD length $simd_{length}$ *if holds:*

- *The root expression is an SIMD expression,*
- *all operations which are used in SIMD expressions are vectorizable with SIMD length* $simd_{length}$*,*
- *for the set of all involved types* T *used as parameters resulting from scalar expressions in SIMD expressions holds*

$$\forall t \in T \setminus \{boolean\} : \textsc{LengthOf}(t) = simd_{length}$$

and

- *all involved SIMD containers have an SIMD length of* $simd_{length}$.

The statement has the SIMD length $simd_{length}$.

Remark. Section 6.4.3 will clarify whether an operation is vectorizable with SIMD length $simd_{length}$.

6.3.1 Loop Setup

If the SIMD statement is vectorizable, the compiler must generate appropriate code in order to setup the loop. This setup is shown in Algorithm 6.6: Basically, it consists of three loops. Two of them—one loop at the beginning, one at the end—do scalar calculations in order to break down the loop boundaries to a multiple of the given SIMD length $simd_{length}$. The loop, which lies in between, is the main loop and is also called *vector loop*. It performs $simd_{length}$ steps simultaneously and the vectorized versions of the used operations are attendingly taken.

Note that bit arithmetic as shown in Section 2.2 can be used to implement the dividable checks and the calculation of the next/previous multiple of $simd_{length}$.

Lemma 6.4. *The loop setup shown in Algorithm 6.6 iterates over* $[j, k[$.

Remark. This means that *index* adopts all values in $[j, k[$.

Proof. The proof is broken down into four cases:

1. $simd_{length} \mid j \wedge simd_{length} \mid k$: As in this case j_s is set to j and k_s is set to k, only the vector loop is executed. It obviously iterates through $[j, k[$.
2. $simd_{length} \nmid j \wedge simd_{length} \mid k$: We have shown in case 1 that the vector loop iterates till $k-1$. Thus this loop iterates over $[j_s, k[$. But in this case j_s is set to the next multiple of $simd_{length}$. The first loop, however, runs from over $[j, j_s - 1[$. The combination of both intervals culminates in $[j, k[$.
3. $simd_{length} \mid j \wedge simd_{length} \nmid k$: Analogously to case 2, it can be shown that the vector loop iterates over $[j, k_s[$ and that the last loop loops over $[k_s, k[$. Both intervals combined yield $[j, k[$ again as expected.
4. $simd_{length} \nmid j \wedge simd_{length} \nmid k$: Combining case 2 and 3 directly shows that in this case the loop iterates over $[j, k[$.

□

Further loop optimizations especially moving loop invariants out of the loop or substituting multiplications with additions[1] can significantly increase the performance of the loop.

[1] This technique is known as *strength reduction*.

Algorithm 6.6 LoopSetup(j, k, S, $simd_{length}$)

$j_s \leftarrow$ **if** $simd_{length} \mid j$ **then** j **else** next multiple of $simd_{length}$
$k_s \leftarrow$ **if** $simd_{length} \mid k$ **then** k **else** previous multiple of $simd_{length}$
ScalarLoop(j, $j_s - 1$, S)
for $i = j_s$ to $k_s - 1$ step $simd_{length}$ **do** ▷ vector loop
    $index \leftarrow (i, \ldots, i + simd_{length} - 1)$
    let E be the set of scalar expressions being direct children of SIMD expressions
    for all scalar values v_l of type t_l used in E **do**
        if $t = boolean$ **then**
            $s \leftarrow simd_{width} \div simd_{length}$
            $t_{uint} \leftarrow$ appropriate *unsigned integer* with size-of(t_{uint}) $= s$
            $t_{vec} \leftarrow simd_{length}$ *packed* t_{uint}
        else
            $(t_{vec}, foo) \leftarrow$ VecType(t, $simd_{length}$) ▷ $foo \equiv simd_{length}$
        end if
        create new value v_{vec_l} of type t_{vec_l}
        $v_{vec_l} \leftarrow$ Broadcast(i, v_{vec_l}, t_{vec})
    end for
    execute S while using
        - the vector versions of all operations in vector expressions and
        - v_{vec_l} instead of v_l
end for
ScalarLoop(k_s, $k - 1$, S)

Algorithm 6.7 ScalarLoop(j, k, S)

for $i = j$ to k **do**
    $index \leftarrow i$
    for all vector values v_{vec_l} of type t_{vec_l} in S **do**
        $v_l \leftarrow$ Extract(i, v_{vec_l}, t_{vec})
    end for
    execute S while using
        - the scalar versions of all operations and
        - v_l instead of v_{vec_l}
    for all vector values v_{vec_l} which were written to in S **do**
        $v_{vec_l} \leftarrow$ Pack(i, v_l, v_{vec_l}, t_{vec})
    end for
end for

If several statements must be applied for the same indices a second kind of loop may be introduced:

```
simd[j, k]
    foo.bar1(v1)
    foo.bar2(v1)
end
```

In some cases a compiler may be able to merge single SIMD statements into such a loop. But since the program semantics of such an transformation can easily change, it is difficult for a compiler to prove this tranformation correct. Thus it is reasonable to give the programmer direct control over the loop body.

6.3.2 SIMD Index

Since SIMD expressions need a common index for all involved SIMD containers (see also Sections 8.2.1 and 8.2.2) the loop boundaries are given in the prefix of an SIMD statement. In some cases it may be handy for the programmer to reference the current index in the case of a scalar loop break down or a packed type holding the current indices in the case of a vector loop. A reserved keyword may be used to reference the current loop index:

```
simd[j, k]: foo.bar(@i)
```

The expression @i is a read-only variable of an appropriate *SIMD length packed integer*. We call this an *SIMD index*. The definition of variable *index* in Algorithms 6.6 and 6.7 show how this can be done.

The thoughtfully reader will notice that the type of a built-in mechanism to reference the current index cannot be the same for all SIMD statements.

One option to deal with this problem is to make the type sensitive to the statement: Internally, an appropriate scalar integer type can be used. In the case of a typical 64 bit CPU this would be an *unsigned quadword*. The language could simply automatically type this index to an $simd_{length}$ *packed unsigned integer* of size $simd_{width}/simd_{length}$. However, the programmer must be careful as especially a large SIMD length yields in a small type range for the resulting type of the index. For instance, $simd_{length} = simd_{width}$ implies a *packed unsigned byte* for the index type. The loop boundaries can easily exceed FF_{16}.

Another solution is the use of manual casts. This is principally the same approach but the program becomes more transparent and errors resulting from too small type ranges become more obvious.

6.4 Vectorization of the Callee Code

The main task of the vectorization is to generate vectorized versions of the functions which are marked by the programmer as SIMD functions. After some general advisements we start with the vectorization of linear programs, then we consider simple branches, afterwards simple loops are discussed and finally we see how any control flow can be vectorized.

6.4.1 Overview

First of all, it is shown that parallel execution bears some principal problems. Consider the following pseudo-code where foo(i) expects an *integer* i:

$a_1 \leftarrow \text{foo}(1)$
$a_2 \leftarrow \text{foo}(2)$

Let us assume foo(i) does I/O operations and it outputs

```
ai bi
```

on the terminal. Hence the output of the pseudo-code would be:

```
a1 b1 a2 b2
```

Now, if we somehow vectorize foo($\vec{i}$) where the parameter $\vec{i}$ becomes a *2 packed integer* we can write the above instructions like this:

$(a_1, a_2) \leftarrow \text{foo}((1, 2))$

Although a_1 and a_2 are calculated correctly the output on the terminal would be:

```
a1 a1 b2 b2
```

Therefore we define pure programs. The definition is a slightly changed variant of the definition found in Wikipedia [2010]:

Definition 6.6 (Pure Program)**.** *A* program *is called* pure *if both these statements about the program hold:*

1. *The program always evaluates the same result value given the same argument value(s). The program result value cannot depend on any hidden information or state that may change as execution proceeds or between different executions of the whole program, nor can it depend on any external input from I/O devices.*

2. *Evaluation of the result does not cause any semantically observable side effect or output, such as mutation of mutable objects or output to I/O devices.*

Vectorization of impure programs may be done nevertheless but the programmer should be aware of the resulting problems. For instance, it just might not matter in which order an I/O device is accessed. This is often the case with debug output and it would make programming tedious if only the release version which does not use debug output is allowed to be vectorizable. As a result we only consider pure programs from now on.

As second example we assume

$a_1 \leftarrow \text{foo}(1)$

terminates but

$a_2 \leftarrow \text{foo}(2)$

does not. Now the parallel execution

$(a_1, a_2) \leftarrow \text{foo}((1, 2))$

will not terminate neither and it is possible that a_1 will never be computed although a_1 gets calculated in the scalar version.

Definition 6.7 (Total Program)**.** *A* program *is called* total *if it terminates for all possible arguments.*

Vectorization of programs which are not total is possible, too. Especially, if the program does not terminate only for some corner cases, which are known by the programmer and simply do not occur in practice, vectorization makes sense anyway. Additionally, in most cases a program which does not terminate is more a programming flaw than intention. However, as with pure programs we only consider total programs as from now, but nothing stops a compiler from vectorizing impure and/or not total programs, too. Because of the halting problem a compiler is not able to prove a program not being total in many cases, anyway.

Definition 6.8 (Correct Program Transformation)**.** *A* program transformation $\mathcal{T}(P) = P_\mathcal{T}$ *of a pure and total program* P *with*

- n *parameters* $p_i^P \in V_P$ *with* type-of$(p_i^P) =: t_i^p$ *and*
- m *results* $r_j^P \in V_P$ *with* type-of$(r_j^P) =: t_j^r$

to a pure and total program $P_\mathcal{T}$ *with*

- n *parameters* $p_i^{P_\mathcal{T}} \in V_{P_\mathcal{T}}$ *with* type-of$(p_i^{P_\mathcal{T}}) \equiv t_i^p$ *and*
- m *results* $r_j^{P_\mathcal{T}} \in V_{P_\mathcal{T}}$ *with* type-of$(r_j^{P_\mathcal{T}}) \equiv t_j^r$

where $i \in N = \{k | 1 \leq k \leq n \wedge k \in \mathbb{N}\}$ *and* $j \in M = \{k | 1 \leq k \leq m \wedge k \in \mathbb{N}\}$ *is* correct *if holds:*

$$\forall i \in N \, \forall d_i = \text{value-of}(p_i^P) = \text{value-of}(p_i^{P_\mathcal{T}}) \in \mathcal{D} :$$
$$P(p_1^P, \ldots, p_n^P) = (r_1^P, \ldots, r_m^P) \equiv (r_1^{P_\mathcal{T}}, \ldots, r_m^{P_\mathcal{T}}) = P(p_1^{P_\mathcal{T}}, \ldots, p_n^{P_\mathcal{T}})$$

Remark. This also implies that $P_{\mathcal{T}}$ is pure and total, too.

Since the program transformation presented in this chapter will also transform the types of the parameters we use small wrappers (see Algorithm 6.8 and 6.9) to show the correctness of the transformation in the sense of Definition 6.8. In the case of $\textsc{WrapScalar}_P$ a scalar vectorizable program P is invoked internally. In the case of $\textsc{WrapVec}_{P_{vec}}$ a vectorized program P_{vec} is called.

Algorithm 6.8 $\textsc{WrapScalar}_P(a_{10}, \ldots, a_{n0}, \ldots, a_{1(simd_{length}-1)}, \ldots, a_{n(simd_{length}-1)})$

for $i = 0$ to $simd_{length} - 1$ **do**
  $(r_{1i}, \ldots, r_{ji}) \leftarrow P(a_{1i}, \ldots, a_{ki})$
end for
return $(r_{10}, \ldots, r_{m0}, \ldots, r_{1(simd_{length}-1)}, \ldots, r_{m(simd_{length}-1)})$

Algorithm 6.9 $\textsc{WrapVec}_{P_{vec}}(a_{10}, \ldots, a_{n0}, \ldots, a_{1(simd_{length}-1)}, \ldots, a_{n(simd_{length}-1)})$

create variables a_i of type $t_i^a = \textsc{VecType}(\text{type-of}(a_{i0}))$ with $i = 1, \ldots, n$
create variables r_i of type $t_i^r = \textsc{VecType}(\text{type-of}(r_{i0}))$ with $i = 1, \ldots, m$
for $j = 1$ to n **do**
  for $i = 0$ to $simd_{length} - 1$ **do**
    $a_j \leftarrow \textsc{Pack}(i, a_{ji}, a_j, t_j^a)$
  end for
end for
$(r_1, \ldots, r_j) \leftarrow P_{vec}(a_1, \ldots, a_k)$
for $j = 1$ to m **do**
  for $i = 0$ to $simd_{length} - 1$ **do**
    $r_{ji} \leftarrow \textsc{Extract}(i, r_{ji}, t_j^r)$
  end for
end for
return $(r_{10}, \ldots, r_{m0}, \ldots, r_{1(simd_{length}-1)}, \ldots, r_{m(simd_{length}-1)})$

Since both $\textsc{WrapScalar}_P$ and $\textsc{WrapVec}_{P_{vec}}$ are obviously pure and total if P and P_{vec} are pure and total we can now define a correct vectorization:

Definition 6.9 (Correct Vectorization)**.** *A* vectorization with SIMD length $simd_{length}$

$$\mathcal{T}_{vec}(P) = P_{vec}$$

of a pure and total program P in strict SSA form is called correct, *if*

- *P is vectorizable with SIMD length $simd_{length}$,*

- *the program transformation* $\mathcal{T}(\textsc{WrapScalar}_P) = \textsc{WrapVec}_{P_{vec}}$ *is correct,*
- *the resulting program is again in strict SSA form,*
- *the transformation can be done in* $\mathcal{O}(n)$ *time where* n *is the number of instructions in the program.*

Remark. This also implies that P_{vec} is pure and total. Note that the running time of $\mathcal{O}(n)$ is optimal. Definition 6.10 clarifies whether a program is vectorizable with an SIMD length $simd_{length}$.

Theorem 6.1. *The transformation of a pure and total program*

$$\textsc{ScalarLoop}(j,\ k,\ S)$$

where S *is some vectorizable statement with SIMD length* $simd_{length}$ *and all needed vectorized operations are correctly vectorized with SIMD length* $simd_{length}$ *in the sense of Definition 6.9 to a program*

$$\textsc{LoopSetup}(j,\ k,\ S,\ simd_{length})$$

is correct.

All vector values exactly fit into $simd_{width}$ *wide registers while the vector loop performs* $simd_{length}$ *many calculations simultaneously in one iteration.*

Proof. By Lemma 6.4 all indices of the original loop are taken into account. The invocation of ScalarLoop in LoopSetup is in fact part of the original program. Hence we must only show that the part of the loop, which is used for the vector loop, is correctly transformed. Since the program is pure and total, the order of the execution does not matter. Now the vector loop performs $simd_{length}$ many calculations simultaneously. It follows from Definition 6.9 that each component is calculated exactly like in the scalar version. Hence the transformation is correct.

Lemma 6.2 implies that all vector values have the size $simd_{width}$ while these values consist of $simd_{length}$ many components. Hence $simd_{length}$ many calculations are simultaneously performed in one iteration of the vector loop. □

Remark. This also shows why Definition 6.9 is reasonable. Now we must show that the vectorization algorithms fulfill this definition.

6.4.2 Prerequisites

In this section we introduce some further constraints to the source program which needs vectorization.

Supported Control Flows

As the vectorization of a program depends on the structure of the control flow we assume for the moment that the high-level language only supports these structures in addition to simple linear control flows:

If-else-construct: Figure 6.1a sketches this construct. It consists of a header (block 1) which dominates the rest. There the control flow divides into an if-block and into an else-block (block 2 and 3). The flow merges again in block 4. Note, that an if-construct i.e., an if-else-construct with a missing else-block (block 3, for instance), is transformed into the given layout by a critical edge elimination pass because the edge $1 \to 4$ is critical if block 3 is missing. In such a case an empty block would be inserted between $1 \to 4$ and this layout would again emerge.

While-loop: Such loops are known from many programming languages (Figure 6.1b) A loop header (block 2) checks an exit condition. Upon exit the loop breaks via the exit edge $1 \to 2$. Otherwise the loop's body (block 3) is executed and the execution jumps back to the header via the back edge $3 \to 2$. It should be pointed out that C style for-loops or Java style for-each loops are principally while-loops.

Repeat-until-loop: This kind of loop (see Figure 6.1c) also known as do-while-loop checks the exit condition at the end of the loop's body (block 2). In this case block 2 is also the header and edge $2 \to 2$ is the back edge whereas edge $2 \to 3$ forms the exit edge. In contrast to while-loops it is guaranteed that at least one iteration is performed. Compilers often transform while-loops into an if-else-construct followed by an repeat-until-loop since the execution of a repeat-until-loop is slightly faster than the execution of a while-loop.

The exact definition of *loops*, *back edges* and *exit edges* are given in Section 6.4.6 while these control flows are precisely defined in Definitions 6.11, 6.12 and 6.13.

These control flows may be freely nested. Hence with support for these control flows we can handle the most common structures available in programming languages. However, other control flows which may occur with the use of **break**, **continue** or **goto**, for instance, are not supported at the moment. We will discuss other techniques later. Furthermore we assume that other passes in the compiler like optimization passes do not transform a program into a forbidden control flow.

Summary

The following algorithms work on an IR of programs while these assumptions are being made:

- The program is pure and total (see Definition 6.6 and Definition 6.7).

- The program uses an IR as defined in Section 2.4.
- The program only uses the control flows presented in Section 6.4.2.
- All critical edges in the CFG have been eliminated (see Section 2.5.1).
- The program is in SSA form (see Section 2.6).
- The program is strict (see Definition 2.1 and Theorem 2.1).
- The program is vectorizable (see Definition 6.10).

For convenience these preconditions are not repeated anymore.

6.4.3 SIMD Length of Programs

First of all, we will check whether a program is vectorizable and compute its SIMD length. The compiler should give an error message if vectorization is not possible. This is shown in Algorithm 6.10.

Algorithm 6.10 CheckProgram(P)

```
simd_length ← 0
for all operands o ∈ O_P do
    tmp ← LengthOf(type-of(o))
    if tmp = −1 then
        return −1
    else if simd_length = 0 then
        simd_length ← tmp
    else if simd_length ≠ tmp then
        return −1
    end if
end for
if simd_length = 0 then
    simd_length ← simd_width
end if
return simd_length
```

Definition 6.10 (Vectorizable Program)**.** *A* program P *is called* vectorizable, *if*

$$\textsc{LengthOf}(P) \neq -1 \ .$$

The associated SIMD length of a vectorizable program P *is defined as* $simd_{width}$, *if* $\textsc{LengthOf}(P) = 0$ *or* $\textsc{LengthOf}(P)$ *otherwise.*

As this approach is very similar to Algorithm 6.1 we waive a correctness proof. Since the only loop depends on the number of operands used in P the running time is $\mathcal{O}(n)$ where n is the number of instructions in P by Lemma 2.1.

Algorithm 6.10 assumes that the type of parameters are not *pointers*. But especially the use of call-by-reference makes use of *pointers*—at least internally. This is not a restriction, however. Only the Algorithm must be slightly adapted:

Each (vectorizable) call-by-reference parameter must be tagged with its original type. As long as the program only uses the pointers in order to get read or write access of the data everything is fine.

6.4.4 Vectorization of Linear Programs

After the compiler has found out that a program is vectorizable the actual vectorization must be performed. First of all, we consider a linear program flow. This means that the program does not contain any *jump instructions*. Since the scalar version is still needed by the compiler a copy must be made while substituting each operand with an appropriate vector operand and replacing each instruction with an appropriate vector instruction of the source program as shown in Algorithm 6.11.

Algorithm 6.11 VecProgram(P)

$simd_{length} \leftarrow$ CheckProgram(P)
if $simd_{length} = -1$ **then**
    return error
end if
create empty new program P_{vec}
for all operands $o \in O_P$ **do**
    if $o \in C_P$ with $o = (c, t)$ **then**
        $C_{P_{vec}} \leftarrow C_{P_{vec}} \cup (c, simd_{length}\ packed\ t)$
    else ▷ $o \in V_P$ with $o = (id, t)$
        $V_{P_{vec}} \leftarrow P_{vec} \cup (id, simd_{length}\ packed\ t)$
    end if
end for
for all instructions i in P in descending order **do**
    create proper vectorized instruction i_{vec} of i
    append i_{vec} to P_{vec}'s instruction list
end for
VecIfElse(P_{vec}, P_{vec}'s starting basic block, $simd_{length}$)
VecLoops(P_{vec}, P_{vec}'s starting basic block, $simd_{length}$)
return P_{vec}

As we show later on the vectorization of if-else-constructs and loops can be performed after the copy. But for the moment we ignore the invocation of VecIfElse and VecLoops.

The algorithm simply states to create a proper vectorized instruction from the source one. Since this depends on the exact instruction set of the IR we give some examples. Although we only discuss programs with a linear control flow at the moment we also show what to do in the case of jump instructions and Φ-functions since VecIfElse and VecLoops, which are presented later, need modified copies:

- Labels can simply be copied. Since a new program is created both labels still are unique in their program.
- Simple arithmetic instructions are substituted by an appropriate SIMD instruction.
- Comparisons are replaced by appropriate SIMD comparisons.
- Calls to other programs are substituted to calls of the vector version of the program.
- More complex instructions which do not alter the linear control flow and do not have a vectorized pendant can be wrapped around via Algorithm 6.8 and 6.9.
- Goto instructions can simply be copied.
- Branch instructions are replaced by an appropriate branch instruction which takes an appropriate *packed integer* as operand.
- A Φ-function is copied while the new vectorized operands are taken as arguments and results.

Figure 6.2 shows an example program and how the altered copy constructed by Algorithm 6.11 (without the invocation of VecIfElse and VecLoops) looks like.

Lemma 6.5. *Algorithm 6.11 is a correct vectorization for a linear program. The resulting program is again in strict SSA form.*

Remark. Note the further preconditions mentioned in Section 6.4.2.

Proof. Since the source program is total and pure the order of execution of calculations which do not depend on each other does not matter. This is the case for each element of the input and result variables (which are all *packed* typed variables or *records* consisting of other *records* or *packed* types) since their calculations do not interfere. Now the order of execution of the calculation for one element is not changed at all and all calculations are in fact unaltered. Hence the vectorization is correct.

Since the new program does neither introduce new variables nor does it eliminate ones and the identifiers are simply copied the new program is also in strict SSA form.

The second loop takes n iteration. The first loop takes k iterations where k is the number of operands in P. With Lemma 2.1 this adds up to a running time of $\mathcal{O}(n)$. □

6.4.5 Vectorization of If-Else-Constructs

From now on, we consider branches. The simplest constellation is an if-else-construct. For now we assume that this layout together with while- and repeat-until-loops are the only control flows supported by the high-level language as already discussed in Section 6.4.2.

Definition 6.11 (If-Else-Construct)**.** *An* If-else-construct *within a program with a CFG* $G = (V, E)$ *consists of*

- *a subgraph* $G_{if} = (V_{if}, E_{if})$ *of* G,
- *a subgraph* $G_{else} = (V_{else}, E_{else})$ *of* G *with* $V_{if} \cap V_{else} = \emptyset$,
- *a header block* $h \in V \setminus (V_{if} \cup V_{else})$ *with* $|succ(h)| = 2$ *and*
- *a merge block* $m \in V \setminus (V_{if} \cup V_{else})$ *with* $|pred(m)| = 2$.

We define:

$$\begin{aligned} H_{if} &:= \{i \to j \mid i \in V \setminus V_{if} \ \wedge \ j \in V_{if}\} \\ H_{else} &:= \{i \to j \mid i \in V \setminus V_{else} \ \wedge \ j \in V_{else}\} \\ M_{if} &:= \{i \to j \mid i \in V_{if} \ \wedge \ j \in V \setminus V_{if}\} \\ M_{else} &:= \{i \to j \mid i \in V_{else} \ \wedge \ j \in V \setminus V_{else}\} \ . \end{aligned}$$

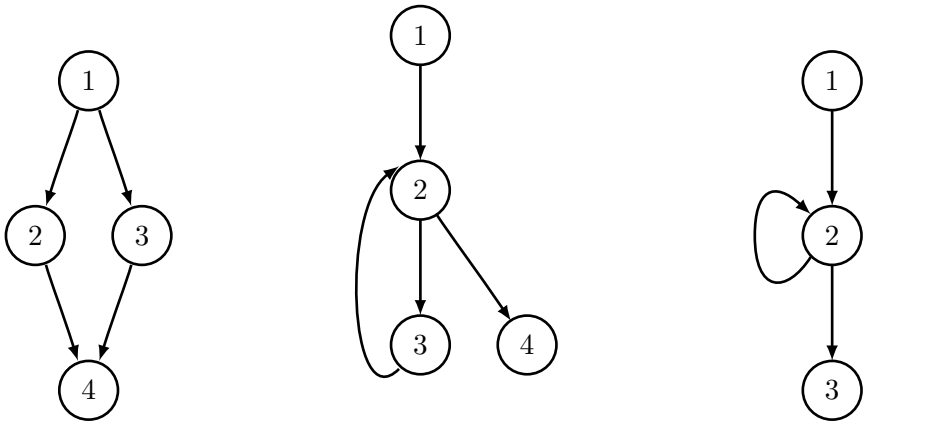
Now it holds

$$\begin{aligned} H_{if} &= \{h \to i_1\} \\ H_{else} &= \{h \to i_2\} \\ M_{if} &= \{e_1 \to m\} \\ M_{else} &= \{e_2 \to m\} \end{aligned}$$

with $i_1, i_2 \in V_{if}$ *and* $e_1, e_2 \in V_{else}$.

Furthermore, we need a way to distinguish branches from loops. Algorithm 6.12 uses the program P, a basic block b and the data structure $\mathcal{D}$ which is presented in Section 6.14 in order to determine the merge block of an if-else-construct if b is a header of such a construct or **nil** otherwise.

Lemma 6.6. *The header of an if-else-construct strictly dominates all nodes in its subgraphs and its merge node.*



(a) If-else-construct (b) While-loop (c) Repeat-until-loop

Figure 6.1: Supported control flow constructs

Algorithm 6.12 IsIfElse(P, b, $\mathcal{D}$)

$branch \leftarrow b$'s last instruction
if b is not a branch instruction **then**
    return nil
end if
▷ check whether b is a header of a while-loop
if $\mathcal{D}[b] \neq$ **nil then**
    return nil
end if
$b_1 \leftarrow s \in succ(b)$
$b_2 \leftarrow s \in succ(b) \wedge s \neq b_1$
▷ check whether one of b's successors is a header of a repeat-until-loop
if $\mathcal{D}[b_1] \neq$ **nil** $\vee\ \mathcal{D}[b_1] \neq$ **nil then**
    return nil
end if
return $df(b_1)$

l_{start}:
  $i_1 \leftarrow 0$
  $a \leftarrow \ldots$
l_{while}:
  $i_2 \leftarrow \Phi(i_1, i_3)$
  $c \leftarrow i_2 < a$
  if c **then** l_{loop} **else** l_{exit}
l_{loop}:
  $\ldots$
  $i_3 \leftarrow i_2 + 1$
  $\ldots$
  goto l_{while}
l_{exit}:
  $\ldots$

(a) Source program in SSA form

l_{start}:
  $\vec{i}_1 \leftarrow \vec{0}$
  $\vec{a} \leftarrow \ldots$
l_{while}:
  $\vec{i}_2 \leftarrow \Phi(\vec{i}_1, \vec{i}_3)$
  $\vec{c} \leftarrow \vec{i}_2 \oslash \vec{a}$
  if $\vec{c}$ **then** l_{loop} **else** l_{exit}
l_{loop}:
  $\ldots$
  $\vec{i}_3 \leftarrow \vec{i}_2 \oplus \vec{1}$
  $\ldots$
  goto l_{while}
l_{exit}:
  $\ldots$

(b) Resulting program

Figure 6.2: Application of Algorithm 6.11

Proof. First, we show that the header h dominates all nodes in its subgraph $G_{if} = (V_{if}, E_{if})$: By Definition 6.11 there is exactly one ingoing edge to G_{if}, namely $h \to i \in V_{if}$. This implies h dominates all $i \in V_{if}$.

Similarly, it can be shown that h dominates all nodes of the other subgraph $G_{else} = (V_{else}, E_{else})$.

Now, there are exactly two edges which are incident to the merge node m: $i \in V_{if} \to m$ and $j \in V_{else} \to m$. Thus $h \prec m$, too. □

Lemma 6.7. *Algorithm 6.12 is correct and runs in constant time.*

Proof. If the basic block b does not end with a branch, b cannot be the header of an if-else-construct (first if-case). The allowed control flows of the high-level language imply for a branch that it either emerged from a while-loop (second if-case), a repeat-until-loop (third if-case) or an if-else-construct (last case).

The merge block m is neither dominated by one of V_{if}'s nor by one of V_{else}'s nodes since V_{if} and V_{else} are disjoint. This with Lemma 6.6 implies that m is in the dominance frontier of all nodes in $V_{if} \cup V_{else}$.

The lookup in the dictionary $\mathcal{D}$ can be implemented in constant time if a hash map is used. Alternatively, each basic block can store a pointer to its loop data structure. We assume that the dominance frontier information is precomputed since it is most likely needed by computing SSA form (which is preconditioned on this spot) and/or other optimization passes. The lookup can be implemented in constant time using hashing. □

Now, we consider once again the program in Figure 2.3b. If this program should be correctly vectorized we must assure the following:

1. All possible control flows are executed. This can be achieved by putting the else block directly after the if block whereas the flow from the if block to the merge block must be eliminated. Instead, the merge block must be executed after the execution of the else block.

2. After altering the control flow it must be guaranteed that the correct values are computed, nevertheless. This can be done by masking all variables in question which has already been discussed in Section 3.3.1.

Figure 6.3 illustrates the transformation: According to the first consideration the CFG is transformed into a linear one. Then the Φ-function becomes substituted by a proper blending operation.

Algorithm 6.13 shows this approach precisely: First, it does a post order walk of the dominance tree. This guarantees that the most nested if-else constructs are transformed first. Then the CFG gets rewired as discussed above. The algorithm also handles if- and else blocks correctly which may consist of other nested structures. At last Φ-functions in the merge block are properly exchanged by masking operations.

Lemma 6.8. *Algorithm 6.11 including the invocation of Algorithm 6.13 is a correct vectorization of a program, which only consists of linear control flows and if-else-constructs. The resulting program is again in strict SSA form.*

Remark. Note the further preconditions mentioned in Section 6.4.2.

Proof. The post order walk of the dominance tree assures that the most nested constructs are vectorized first. Thus we transform from inside out.

The rewiring (correct by Lemma 6.7) guarantees that all possible control flows are executed. Since the program is in SSA form no values get overwritten when executing the else-subgraph after the if-subgraph. Theorem 2.1 implies that all variables computed in either the if-subgraph or the else-subgraph which are needed in the merge block or afterwards are used in arguments of a Φ-function at the start of the merge block. By Lemma 3.1 we can select the correct values via Algorithm 3.1.

Since no new definitions are introduced the transformed program again is in strict SSA form.

Algorithm 6.13 VecIfElse(P, b, $simd_{length}$)

▷ do a post order walk of the dominance tree
for all $\tilde{b} \mid \tilde{b} = idom(b)$ **do**
    VecIfElse(P, $\tilde{b}$, $simd_{length}$)
end for

▷ *next* is the block where the control flow merges
next ← IsIfElse(P, b)
if *next* = **nil then**
    return
end if

branch ← b's last instruction which is a branch instruction
ifChild ← *branch*'s if target
elseChild ← *branch*'s else target

▷ *pred*(*next*) has exactly two elements
lastIf ← $p \in pred(next)$
lastElse ← $p \in pred(next) \wedge p \neq lastIf$

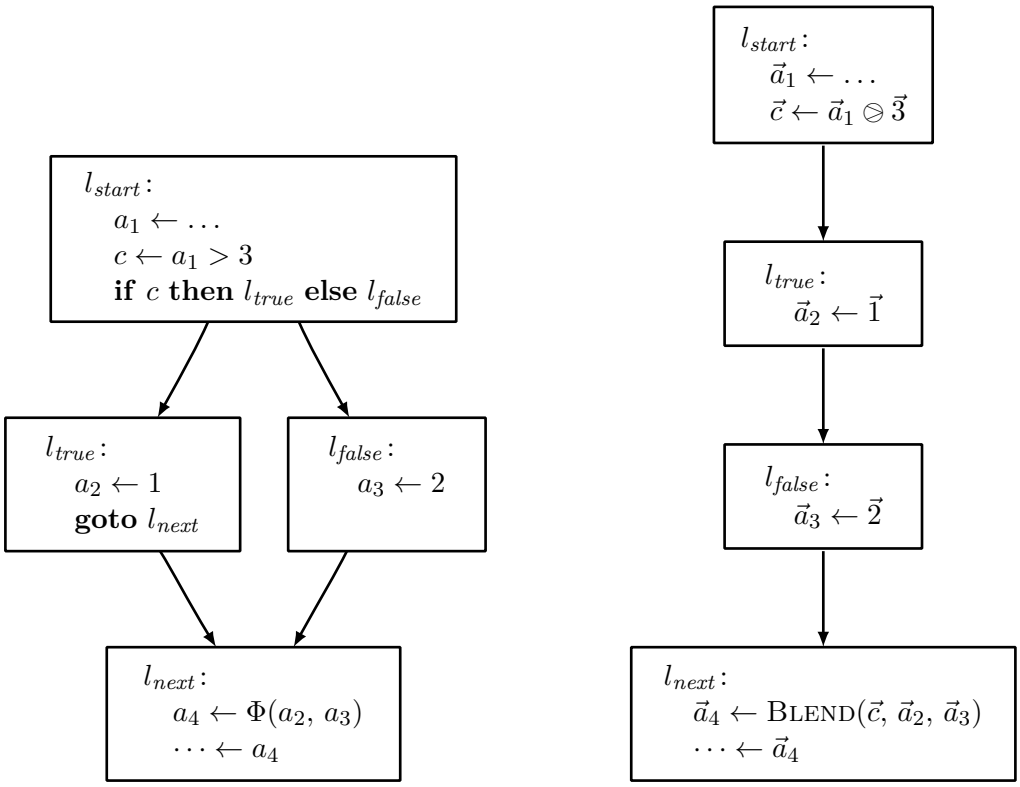
▷ check whether this choice was correct and swap if necessary
if *ifChild* $\prec$ *lastElse* **then**
    swap(*lastIf*, *lastElse*)
end if

▷ rewire CFG
erase edge $b \rightarrow elseChild$
erase edge $lastIf \rightarrow next$
add edge $lastIf \rightarrow elseChild$

if *lastIf*'s or *lastElse*'s last instruction is a goto instruction **then**
    remove goto instructions
end if

$\vec{c} \leftarrow arg(branch)$

for all Φ-functions $phi : \vec{a} = \Phi(\vec{a}_{if}, \vec{a}_{else})$ in *next* **do**
    substitute *phi* with this instruction
        $\vec{a} \leftarrow$ Blend($\vec{c}$, $\vec{a}_{if}$, $\vec{a}_{else}$)
end for



(a) If-else-construct (b) Vectorized if-else-construct

Figure 6.3: Vectorization of an if-else-construct

Assuming, that the program just consists of l labels, p Φ-functions , some branches and some definitions that we do not even take into account, the post order walk takes l steps while p substitutions must be made. With $n > l + p$ instructions and Lemma 6.7 follows a running time of $\mathcal{O}(n)$. □

Remark. The running time could perhaps be estimated better but this is not necessary because of Algorithm 6.11 and Definition 6.9.

Merging Basic Blocks

After the application of the last algorithm some basic blocks can be merged into a single one. The nodes l_{start}, l_{true}, l_{false} and l_{next} in Figure 6.3b can be merged into a single node.

This has several advantages for following passes in the compiler: Some algorithms just consider a single block at a time and do not bother to combine the results to a better

solution. Some register allocation algorithms for just-in-time compilers are an example. For many algorithms a global combining is expensive. Hence it is better to minimize the need for it. Other algorithms create better solutions if the need for global combining is kept at a minimum.

6.4.6 Vectorization of Loops

First of all, we must define what exactly a loop is. We only consider *natural loops* here since we assume a reducible CFG (see Section 6.4.7). These loops are defined via *back edges*. An edge $i \to j$ is called *back edge* if $j \preceq i$. Additionally, we call an edge $i \to j$ an *exit edge* of some loop L if $i \in L$ and $j \notin L$.

Definition 6.12 (While-Loop)**.** *A* while-loop *within a program with a CFG $G = (V, E)$ consists of*

- *a subgraph $G_{body} = (V_{body}, E_{body})$ of G,*
- *a loop header $h \in V_{body}$ where $\forall i \in V_{body} : h \preceq i$,*
- *exactly one back edge $n \in V_{body} \to h$ where $\forall i \in V_{body} : i \preceq n$ and*
- *exactly one exit edge $h \to i \in V \setminus V_{body}$.*

By just adjusting the placement of the exit edge we can define a repeat-until-loop:

Definition 6.13 (Repeat-Until-Loop)**.** *A* repeat-until-loop *within a program with a CFG $G = (V, E)$ consists of*

- *a subgraph $G_{body} = (V_{body}, E_{body})$ of G,*
- *a loop header $h \in V_{body}$ where $\forall i \in V_{body} : h \preceq i$,*
- *exactly one back edge $n \in V_{body} \to h$ where $\forall i \in V_{body} : i \preceq n$ and*
- *exactly one exit edge $i \in V_{body} \to j \in V \setminus V_{body}$ where $\forall n \in V_{body} : n \preceq i$.*

Remark. Since all basic blocks including the header node are part of the loop's body we also sometimes just denote the set V_{body} as the loop. This was already used in the definition of the exit edge.

More complex loop structures are possible but for now we just assume that the high-level language only supports these ones.

Detecting Loops

The next obstacle is to identify all loops in a program. A dictionary $\mathcal{D}$ is used in this place which maps a loop's header to a pointer to $\mathcal{L}$. This data structure holds the following information:

- The loop's *header* basic block,
- a set *body* which holds all basic blocks including the header which belongs to the loop,
- a set *back* which stores all back edges to the header and
- a set *exit* which stores all exit edges out of a loop.

Algorithm 6.14 detects all natural loops within a program and not only while- and repeat-until-loops. It fills the data structure and the dictionary. If several back edges lead to the same header these loops are merged into a single one. The algorithm is a slightly enhanced version of the one presented by Aho et al. [pg. 603–604 1986].

The algorithm sometimes does multiple insertions of the same element into the same set which is not a problem, however. Furthermore, we assume that this loop detection is done anyway in a compiler. Hence it does not increase the running time of the vectorization. As already mentioned in Section 6.4.5 the lookups in the dictionary $\mathcal{D}$ can be done in constant time.

An important property of a loop's header is that it dominates all basic blocks within this loop.

Pre-Headers

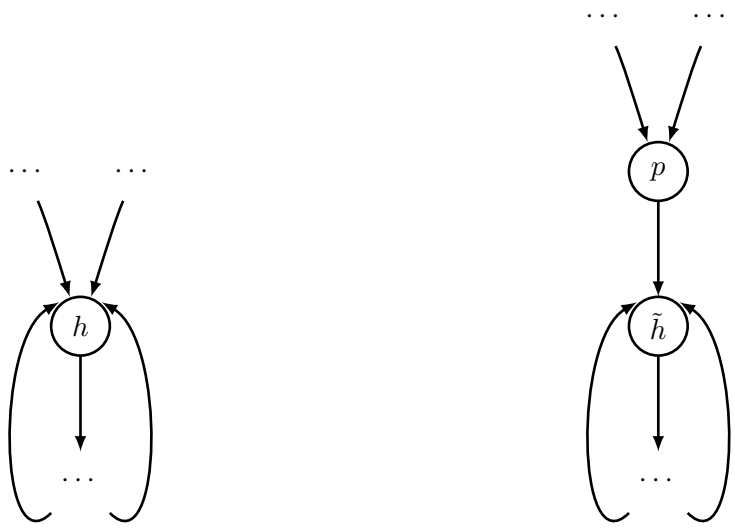
It is common practice for many loop optimizations to split a loop's header into the actual header and a pre-header as shown in Figure 6.4: All predecessors of a header h of a loop L which enter L from outside of L (see Figure 6.4a) are put as predecessors to a new node p (see Figure 6.4b). All predecessors which come from inside L are put to the new header $\tilde{h}$. A new edge $p \to h$ is inserted. All ordinary instructions of h are put into $\tilde{h}$ while all Φ-functions of h are put into p. This allows an optimization to put an instruction just before the loop.

Vectorization of While-Loops

In contrast to if-else-constructs the control flow cannot be transformed into a linear one since the loop boundaries are not necessarily known at compile time. Thus the strategy is that all possible control flows are executed. This also means that not all definitions are valid. Invalid definitions must be properly masked.

Algorithm 6.14 FindLoops(P)

```
create empty dictionary 𝒟
for all edges i → j in the CFG of P do
    if j ≺ i then ▷ i → j is a back edge
        if j ∈ 𝒟 then
            loop ← 𝒟[j]
        else
            loop ← new ℒ
            loop.header ← j
        end if
        loop.back.insert(i → j)
        loop.body.insert(i)
        loop.body.insert(j)
        let 𝒮 be an empty stack
        𝒮.push(i)
        while 𝒮 is not empty do
            n ← 𝒮.pop()
            for all p ∈ pred(n) do
                loop.body.insert(p)
                𝒮.push(p)
            end for
        end while
        𝒟[j] ← loop
    end if
end for
for all loop ∈ 𝒟 do
    for all edges i → j in the CFG of P with i ∈ loop.body ∧ j ∉ loop.body do
        loop.exit.insert(i → j)
    end for
end for
return 𝒟
```



(a) A loop L with a header node h

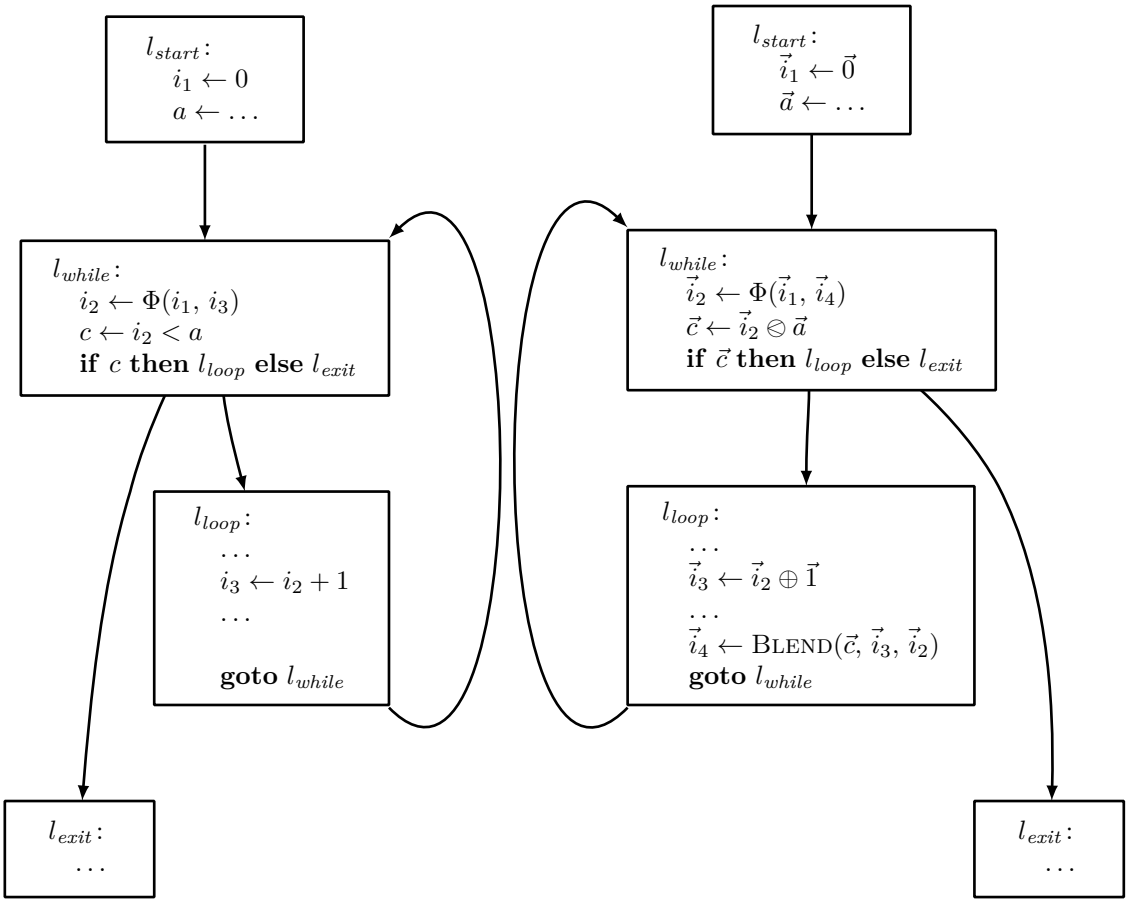
(b) The former header h of L is split into a new header $\tilde{h}$ and into a pre-header p

Figure 6.4: Insertion of a pre-header

Figure 6.5a shows the CFG of the program in Figure 6.2a. After the simple linear vectorization (see Figure 6.2b) the exit condition $\vec{c}$ is only taken if $\vec{c} = \vec{0}$. Hence the loop is executed until the condition is met for all elements of $\vec{c}$. Since this behavior is not desirable we must negate the condition and swap the targets of the branch instruction if the loop's body is executed in the else-case and the exit condition is taken in the if-case. Algorithm 6.15 shows how this can be done.

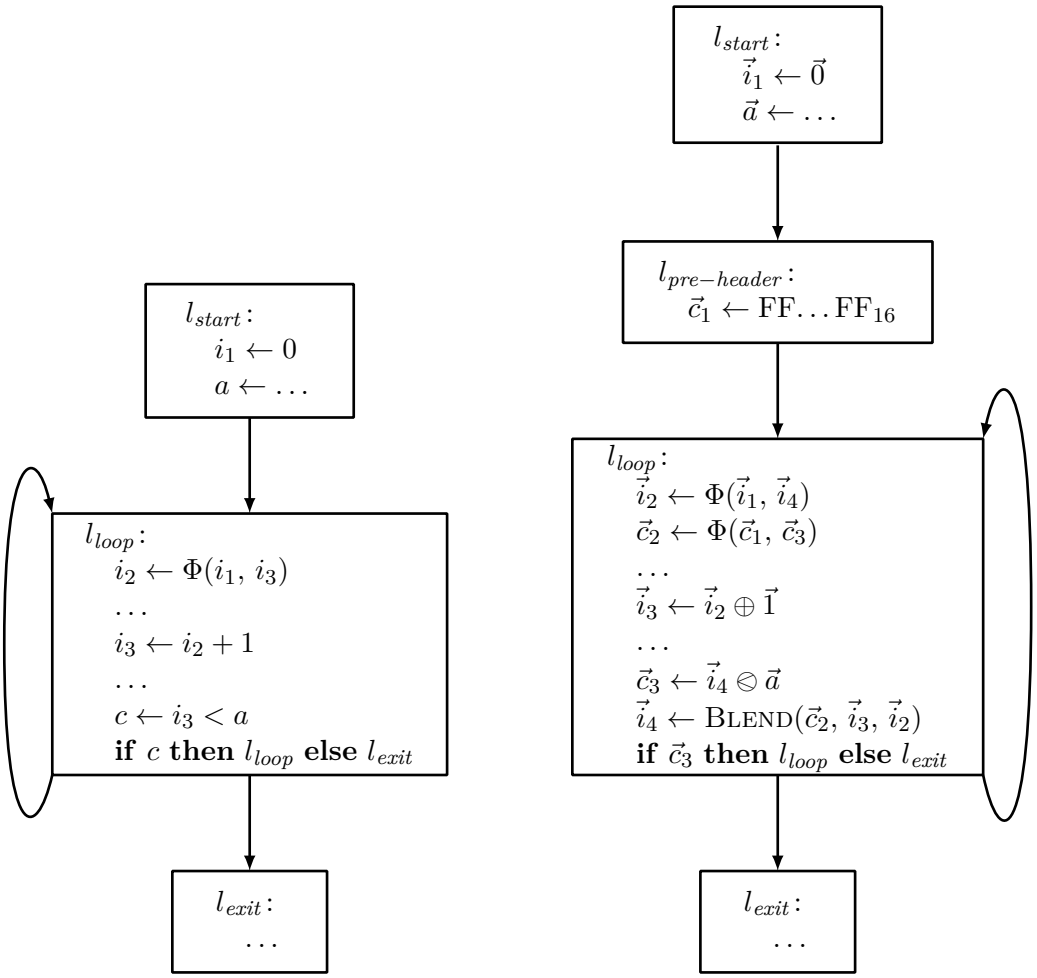
Now this condition can be used as mask. In the case that $\vec{c}[i] = 0$ all definitions of the i^{th} element which also live outside the loop are not valid. The Φ-functions in the loop's header indicate which variables must be masked. Therefore if we put a Blend at the end of the loop's body just before the jump back to the header and update this new definition in its associated Φ-function the loop is properly vectorized.

Figure 6.5b shows the vectorized version of the program in Figure 6.5a: The Φ-function shows us which variable must be masked—i_3 in this case. As discussed above we add a new definition i_4 with a masking sequence using $\vec{c}$ as mask. At last we must substitute this new definition in the Φ-function.



(a) A simple while-loop (b) Vectorized version of the while-loop

Figure 6.5: Vectorization of a while-loop



(a) A simple repeat-until-loop

(b) Vectorized version of the repeat-until-loop

Figure 6.6: Vectorization of a repeat-until-loop

Algorithm 6.15 TwistBranch(P, b)

branch ← b's last instruction which is a branch instruction
op ← *arg*(*branch*)
prepend before *branch* a new instruction i_{not}:
    tmp ← **not** *op*
replace *arg*(*branch*) with *tmp*
swap targets of *branch*
▷ keep track of def-use information
remove *branch* as a use of *op*
add *tmp* as a use of *op*
add *tmp* as a definition at i_{not}
add *branch* as a use of *tmp*

Vectorization of Repeat-Until-Loops

The vectorization of repeat-until-loops is very similar. Figure 6.6a shows a simple repeat-until-loop. The exit condition $\vec{c}$ can be used as mask, too. But since its definition is likely only just before the branch we must add a definition

$$\vec{c} \leftarrow \mathrm{FF}\ldots\mathrm{FF}_{16}$$

which dominates the loop's body. As shown above the pre-header is a good place.

The rest of the vectorization can just be done as for the vectorization of while-loops but since a new definition for $\vec{c}$ has been introduced SSA form must be reconstructed: First, a Φ-function must be inserted which either selects the definition in the pre-header or the last definition in one iteration. In total there are three definitions of $\vec{c}$ now. These ones and all former uses of $\vec{c}$ must be properly renamed. Finally, the complete vectorized version of the loop is constructed as shown in Figure 6.6b.

Combining the Vectorization of While- and Repeat-Until-Loops

With the findings of the last two sections we can construct a combined solution as shown in Algorithm 6.16.

Lemma 6.9. *Let lastBB* → *b be the back edge and branchBB* → *i the exit edge of a while- or repeat-until-loop L. Now each variable, which is updated in one iteration of the loop (i.e., its value is needed in the next iteration), is guarded by a* Φ*-function*

$$a \leftarrow \Phi(a_{pre-loop}, a_{loop})$$

in b, where $a_{pre-loop}$ *is defined in a basic block which strictly dominates b, and* a_{loop} *is the last definition of the value in one iteration.*

Algorithm 6.16 VecLoops(P, b, $simd_{length}$)

▷ do a post order walk of the dominance tree
for all $\tilde{b} \mid \tilde{b} = idom(b)$ **do**
    VecLoops(P, $\tilde{b}$, $simd_{length}$)
end for
$loop \leftarrow \mathcal{D}[b]$
if $loop \neq$ **nil then** ▷ b is a loop with header $b \equiv loop.header$
    $branchBB \leftarrow i$, with $(i \rightarrow j) \in loop.exit$
    $lastBB \leftarrow i$, with $(i \rightarrow j) \in loop.back$
    $branch \leftarrow branchBB$'s last instruction which is a branch instruction
    ▷ revert condition if necessary
    if $branch$'s else target $\in loop.body$ **then**
        TwistBranch(P, $branchBB$)
    end if
    $\vec{c} \leftarrow arg(branch)$
    if $\vec{c}$'s definition is not in the loop **then**
        return ▷ this either a loop executed never, exactly once or infinitely
    end if
    $\vec{m} \leftarrow \vec{c}$
    ▷ ensure that $\vec{c}$'s definition dominates the loop and reconstruct SSA form
    if $b \neq branchBB$ **then**
        insert at $loop$'s pre-header:
            $\vec{c}_1 \leftarrow \mathrm{FF}\ldots\mathrm{FF}_{16}$
        insert just at the beginning of b
            $\vec{c}_2 \leftarrow \Phi(\vec{c}_1, \vec{c}_3)$
        rename $\vec{c}$'s former definition to $\vec{c}_3$
        substitute each former use of $\vec{c}$ with its dominating new definition
        alter def-use information of $\vec{c}$, $\vec{c}_1$, $\vec{c}_2$ and $\vec{c}_3$ accordingly
        $\vec{m} \leftarrow \vec{c}_2$
    end if
    for all Φ-functions $phi : \vec{a} = \Phi(\vec{a}_{pre-loop}, \vec{a}_{loop})$ in b **do**
        prepend before $lastBB$'s jump back
            $\vec{a}_{new} \leftarrow$ Blend($\vec{m}$, $\vec{a}_{loop}$, $\vec{a}$)
        substitute phi's argument $\vec{a}_{loop}$ with $\vec{a}_{new}$:
            $\vec{a} = \Phi(\vec{a}_{pre-loop}, \vec{a}_{new})$
    end for
end if

Proof. Since b is the loop's header an update of a value without a strictly dominating definition would render a program not being strict. Hence it exists a definition of $a_{pre-loop}$ which strictly dominates the loop. Because the program is in SSA form such a value must be guarded by the stated Φ-function. By Theorem 2.1 a_{loop} must be the last definition of the value in one iteration. □

Lemma 6.10. *Algorithm 6.11 including the invocation of Algorithms 6.13 and 6.16 is a correct vectorization of a program. The resulting program is again in SSA form.*

Remark. Note the further preconditions mentioned in Section 6.4.2.

Proof. Since all if-else-constructs have been eliminated (see Lemma 6.8) the only branches left result from either while- or repeat-until-loops. The post order walk of the dominance tree again guarantees that the most nested constructs are vectorized first. The invocation of Algorithm 6.15 ensures that the exit condition only holds if all elements of the condition $\vec{c}$ are *false*.

Values which are not needed in the next iteration do not need a further consideration. By Lemma 6.9 such a value is guarded by a Φ-function

$$\vec{a} \leftarrow \Phi(\vec{a}_{pre-loop}, \vec{a}_{loop}) \ .$$

Now only the elements of the values where $\vec{c}$ is *true* may be updated, by masking with an

$$\vec{a}_{new} \leftarrow \textsc{Blend}(\vec{c}, \vec{a}_{loop}, \vec{a})$$

(correct by Lemma 3.1), putting this blend just before the jump back in *lastBB* (correct by Lemma 6.9) and updating this new definition in the Φ-function's argument.

The definition of $\vec{c}$ strictly dominates *lastBB* in the case of while-loops since it is already live in the header b and $b \preceq lastBB$. In the case of repeat-until-loops $\vec{c}$ is only defined later on, otherwise a loop which is executed never, exactly once or infinitely emerges and this case is intercepted by the algorithm. Thus a definition and an appropriate Φ-function must be inserted. Updating all former uses and its former definition leaves the program again in strict SSA form.

With l labels, p Φ-functions and k other definitions involved in the loop the post order walk traverses all l labels, all p Φ-functions are used and not more than ck other instructions need updates while $c \in \mathbb{N}$ is a constant. With Lemma 2.1 linear running time results. □

6.4.7 Vectorization of More Complex Control Flows

So far, we have only considered vectorization of control flows discussed in Section 6.4.2. This section introduces some ideas in order to vectorize other constructs, too. This section is not as worked out as the previous ones since more research is required to give

a solid general purpose algorithm. Instead of having algorithms which can vectorize every thinkable control flow the following two options make sense, too:

- If the compiler discovers a forbidden control flow it may simply emit an error message. This is particularly reasonable as the more complex the control flow becomes the less efficient becomes the vectorization.[2]
- The control flow in question can be wrapped around with Algorithms 6.8 and 6.9. Hence no vectorization is necessary but a performance gain is not achieved in this case, of course. Algorithms 6.8 and 6.9 even introduce an overhead rendering this portion of the code even less efficient than a pure scalar version. A warning by the compiler should be emitted in this case in order to inform the programmer about this issue.

Reducible Control Flow Graphs

The technique described in this section has already been presented by Hecht and Ullman [1972]. A CFG is called *reducible* if the CFG can be reduced to a single node with successive application of these two transformations:[3]

$\mathcal{T}_1$: Remove all edges $b \to b$.

Figures 6.7a and 6.7b sketch this situation: The edge $1 \to 1$ gets erased. When merging more and more nodes with transformation $\mathcal{T}_2$ this transformation takes care of removing all back edges.

$\mathcal{T}_2$: For all basic blocks b, for which holds $|pred(b)| = 1$ and $pred(b) = p$, delete b and make all successors of b successors of p.

This means that the edge $p \to b$ does not exist anymore. Note, that an edge $b \to p$ would be changed to $p \to p$ which can be eliminated by $\mathcal{T}_1$ again. Figures 6.7c and 6.7d show an example: Here, block 2 gets consumed by block 1.

If the reduction reaches a point where neither of these two transformations is possible, the CFG is called *irreducible*. Irreducible CFGs are known to have a somewhat weird control flow. It can be shown that the use of many common programming constructs like if-else, while-loops, repeat-until-loops etc. always produces a reducible CFG. Some languages are even guaranteed to produce reducible CFGs. Even the use of **goto** results

[2] This is only a rule of thumb but one has to think that even in the case of simple if-else-constructs the principal performance gain is roughly halved as both the if- and the else-branch must be executed in all cases. In loops this situation becomes even worse because the number of iterations is equal to the highest iteration number involved.

[3] Note, that it does not make sense to actually apply these transformations on a program. Its sole use is just to test a CFG for reducibility.

Algorithm 6.17 SplitNode(b)

for all but one $p \in pred(b)$ **do**
  $\tilde{b} \leftarrow b$
  for all $s \in succ(b)$ **do**
    add edge $\tilde{b} \rightarrow s$
  end for
  erase edge $p \rightarrow b$
  add the edge $p \rightarrow \tilde{b}$
end for

in reducible CFGs in most cases. Only the use of very uncommon jumps, like jumping from the middle of a loop under a certain condition directly into the middle of another loop, produces an irreducible CFG. But since the use of the **goto** is discouraged anyway [see Dijkstra, 1968][4] such cases should be rare in practice.

When merging more and more nodes of an irreducible CFG with $\mathcal{T}_1$ and $\mathcal{T}_2$ a graph emerges so that neither of these transformations can reduce this graph any further. Each node of this graph represents one or more nodes of the original CFG. The graph can be made reducible again with the application of this transformation:

$\mathcal{T}_3$: Find a block b with $|pred(b)| > 1$. If more than one node fulfills this property the node which represents the most nodes of the original CFG is chosen. Then Algorithm 6.17 is applied via SplitNode(b).

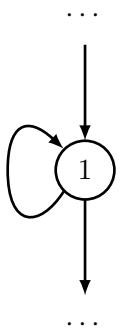
Figures 6.7e and 6.7f illustrate an example: By copying node 2 to a node $\tilde{2}$, rearranging the edge $3 \rightarrow 2$ to $3 \rightarrow \tilde{2}$ and copying the other edge to $\tilde{2} \rightarrow 3$ the CFG is made reducible again.

With successive application of $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$, while the latter one must be appropriately applied on the original CFG as well, the CFG becomes reducible. This technique is known as *node splitting*. Unfortunately, node splitting can result in an exponential increase of the involved nodes in the worst case. But as already outlined above the need for node splitting should be rare anyway and most likely emerges from bad programming habits. We can classify all edges in a reducible CFG into two disjoint sets:

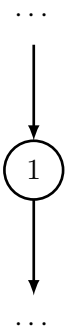
Back Edges: These edges, where the head dominates the tail, have already been defined.

Forward Edges: All edges which are not back edges form a directed acyclic graph where each node can be reached from the start node.

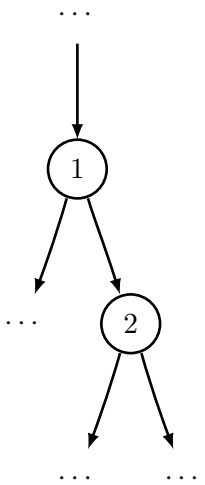
[4] Many programmers do not agree with this statement, however. See http://kerneltrap.org/node/553/2131 for a discussion of this topic.



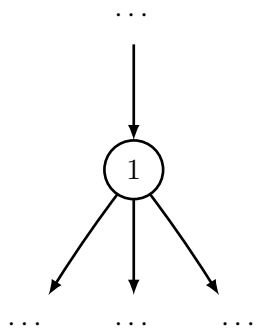
(a) Before the application of $\mathcal{T}_1$



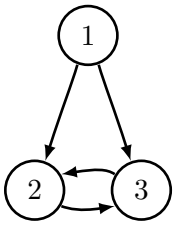
(b) After the application of $\mathcal{T}_1$



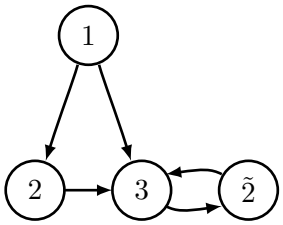
(c) Before the application of $\mathcal{T}_2$



(d) After the application of $\mathcal{T}_2$



(e) Before the application of $\mathcal{T}_3$ (Node Splitting)



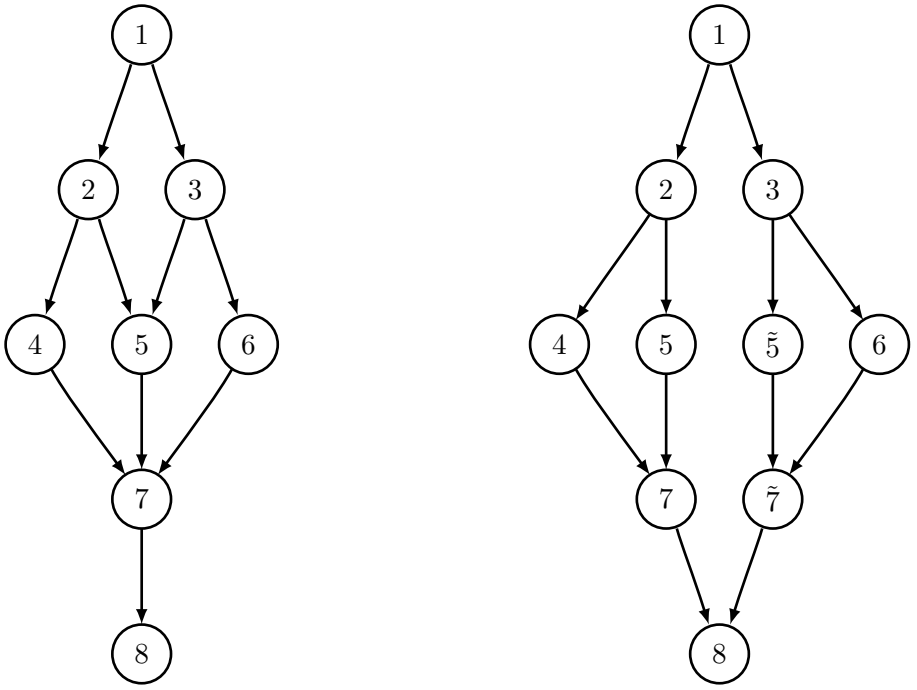
(f) After the application of $\mathcal{T}_3$ (Node Splitting)

Figure 6.7: Program transformations $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$

This also means that there are no other weird edges anymore which can jump from some point to virtually anywhere.

Not Nesting If-Else-Constructs

We now consider a subgraph of a reducible CFG while this subgraph only consists of forward edges which violate Definition 6.11 as shown in Figure 6.8a. Such layouts can emerge when checking conditions which do not nest and making use of **goto** in order to prevent writing the same code again.



(a) Not nesting if-else-constructs

(b) The if-else-constructs nest again

Figure 6.8: Nest arbitrary forward edges

Now, all nodes which violate the property of Definition 6.11 can be copied so the resulting layout nests again as shown in Figure 6.8b. There three if-else-constructs emerge: One consisting of blocks 2, 3, 4 and 7, a second one consisting of blocks 3, $\tilde{5}$, 6 and $\tilde{7}$, and a third one which is formed by the header block 1, the merge block 8 and the other two if-else-constructs as if- and else-structure, respectively.

The nodes can be copied with Algorithm 6.17, too.

Vectorization of Complex Loops

We have already discussed how to vectorize graphs and subgraphs which only consist of forward edges. Now we must take back edges into account. We have already shown how to vectorize the special cases of while- and repeat-until-loops. As already said this topic still needs more research but some general guidelines how this works are given anyway.

If we now allow one or more back edges and random exit edges[5] in a loop we can still vectorize such a construct. Therefore we convert all back edges except the outermost one into exit edges. If there exist several options, an arbitrary one is simply chosen as "outermost" one. The conversion can be achieved by doubling the loop and rewriting the **continue** as **break** as shown in Figure 6.9. If other back edges are used which may emerge from labeled loops and labeled **continue** as known from Java, for instance, these jumps must be properly adapted.

```
while cond1 do
    if cond2 then
        continue
    end if
end while
```

(a) A loop with an additional continue condition

```
while cond1 do
    while cond1 do
        if cond2 then
            break
        end if
    end while
end while
```

(b) The continue has been eliminated

Figure 6.9: Transforming continue conditions to exit conditions

Finally, only exit conditions must be considered:

- First of all, all exit conditions must be rewritten via Algorithm 6.15 so that the exit condition holds in the case that all elements are *false*.
- All other instructions which are executed upon exit before the jump are copied to a special exit code block after the loop.
- All definitions dominated by an exit conditions must be masked properly as shown in Algorithms 6.13 and 6.16.
- All exit condition masks must either be defined in the loop's header or initialized with $\mathrm{FF}\ldots\mathrm{FF}_{16}$ in the pre-header so that these masks dominate all uses in the loop's body.
- Upon exit all exit condition blocks must be executed while masking all definitions with the negated condition.

[5] Exit edges are not a third kind of an edge class, they are just special forward edges.

6.5 Summary

We have shown how data structures must be arranged in order to be useful in an SIMD context. Then the necessary loop setup has been presented in order to iterate over such data structures. Each iteration fetches/writes complete SIMD blocks which are used as in- and out-values for vectorized functions. These functions are automatically generated by the compiler. First, we have limited the supported control flows of the source function to if-else-constructs, while- and repeat-until-loops. Theorem 6.1 summarizes the whole achievement. Basic prerequisite is that all involved types have compatible SIMD lengths.

Additionally, guidelines are given how more complex control flows can be vectorized, too. Section 8.1 deals with some ideas in order to relax the type constraints.

7 Implementation and Evaluation

7.1 The Swift Compiler Framework

The Swift Compiler Framework hosted on BerliOS[1] is a research project of the author which currently consists of a language independent middle-end, a back-end for the x64 architecture and a front-end for an experimental language called *Swift*. The framework is mostly written in C++ with about 18 000 source lines of code (excluding comments and empty lines).

Notable features of the framework are an SSA based IR including an SSA transformation pass, a back-end independent register allocator[2] and a partial implementation of the vectorization algorithms discussed in the last chapter. The latter one suffices to build some benchmarks. These are used to compare the performance of SIMD containers with traditional C/C++ style AoS.

7.2 Implementation Details

In order to make the benchmarks as fair as possible some facts about the implementation in the Swift Compiler Framework must be considered:

- The IEEE-754 standard is not correctly implemented as comparisons involving NaNs should per definition yield *false*. Actually, the parity flag, which is set in the case that a NaN was involved, must be checked after a comparison and the result register must be adjusted. The implementation ignores this at the moment. GCC's option `-ffinite-math-only` delivers the same behavior.

- The x64 back-end of the Swift Compiler Framework manages stack frames just with the stack pointer (%**rsp**) which leaves the frame pointer register (%**rbp**)[3] as an additional general purpose register. The frame pointer is usually used for debugging—especially getting backtraces. In contrast to the x86 standard the x86-64 ABI [see Matz et al., 2010, chap. 3] for GNU/Linux and several BSD variants do not need this register anymore. Instead it uses a more advanced technique

[1] See `http://swiftc.berlios.de`.

[2] The register allocator with optimal graph coloring including the coalescing optimization presented by Hack [2007] has been implemented.

[3] The *frame pointer* is also known as *base pointer* hence the abbreviation %**rbp**.

with a so-called `.eh_frame`.[4] Thus the option `-fomit-frame-pointer` which is normally used to get a free register with the cost of making debugging harder is redundant for x64. The Swift Compiler Framework does currently not support the generation of the `.eh_frame`. However, building this frame does not introduce a runtime overhead and just makes the executable slightly larger. With the option `-fno-dwarf2-cfi-asm` the generation of this data structure can be observed in a more transparent way.

- The framework currently lacks many common optimizations like common subexpression elimination, constant propagation, dead code elimination or instruction scheduling. Therefore the Swift programs are carefully written so that the resulting assembly code is comparable to GCC's output. For the C++ programs this is not necessary as GCC features many optimization passes. Unfortunately, even this act leaves room for several optimizations in the Swift compiler's output.

- Function inlining i.e., substituting a function call with its definition, is a common technique which is frequently used by GCC, too. Since the Swift Compiler Framework currently also lacks such an optimization the C++ implementations are put into different files. This prevents GCC from inlining a function.

The C++ versions were compiled with `-ffinite-math-only` and `-O2` or `-O3`.[5]

7.3 Benchmark

As example computations several operations have been implemented with the hints of the previous section in mind. As integer support is a work in progress most benchmarks make use of *singles* and *doubles*. All benchmarks use one or two input data streams in order to calculate a result stream. These are the tested computations:

vecadd Two streams of 3-D vectors are added up and written to a result container. This has been tested with *singles*, *doubles*, *bytes*, *words*, *doublewords* and *quadwords*. All other benchmarks just use *singles* and *doubles*

veccross Two streams of 3-D vectors used to determine a third stream of vectors by calculating a cross product.

matmul A stream of 3-D vectors is multiplied by a stream of 3×3 matrices.

[4] See http://blogs.sun.com/eschrock/entry/debugging_on_amd64_part_one for more information about this issue.

[5] Refer to the GCC manual (available online at http://gcc.gnu.org/onlinedocs/) to lookup which options are enabled with `-O2` and `-O3`.

ifelse The implementations of the *single* version can be seen in Listings 7.1 and 7.2, respectively. The *double* version looks almost the same.

SIMD containers and arrays with 80 000 000 elements were used.

```
Vec3 Vec3::ifelse(const Vec3& v) {
    Vec3 result;









    if (v.z < 0) {
        result.x = -v.x;
        result.y = -v.y;
        result.z = -v.z;
    } else {
        result.x = v.x * v.x;
        result.y = v.y * v.y;
        result.z = v.z * v.z;
    }


    return result;
}
```

Listing 7.1: C++ implementation of **ifelse** with *singles*

```
simd routine ifelse(Vec3 v) \
    -> Vec3 result
    real resultx
    real resulty
    real resultz
    real vx = v.x
    real vy = v.y
    real vz = v.z
    if vz < 0.0
        resultx = -vx
        resulty = -vy
        resultz = -vz
    else
        resultx = vx * vx
        resulty = vy * vy
        resultz = vz * vz
    end
    result.x = resultx
    result.y = resulty
    result.z = resultz
end
```

Listing 7.2: Swift implementation of **ifelse** with *singles*

Note, how similar the implementations in Listings 7.1 and 7.2 are. The Swift implementation is a bit unconventional as already pointed out above. However, if appropriate optimization passes are made this implementation could be made even more akin to the C++ one without performance penalties. In fact, the constructs presented in this thesis can be implemented within a C++ compiler as well.

Listing 7.3 shows the assembly language output generated by GCC while the above stated options were used to compile the input file. On the other hand Listing 7.4 shows the output of the Swift Compiler. Some labels which are artifacts from prior IRs are left out for better readability. The constants `.LC1` and `.LCSPS`, respectively, are proper masks used with a bitwise XOR in order to implement the negate operation. We can see that the output of the Swift Compiler can be further optimized:

- The adjustment of the stack pointer register (%**rsp**) which is done for alignment reasons is actually not needed here since neither local stack space is needed nor

```
    movss   8(%rdi), %xmm1
    comiss  .LC0(%rip), %xmm1
    jb  .L9
    movss   (%rdi), %xmm0
    mulss   %xmm1, %xmm1
    movss   4(%rdi), %xmm2
    mulss   %xmm0, %xmm0
    mulss   %xmm2, %xmm2
.L4:
    movss   %xmm2, -20(%rsp)
    movss   %xmm0, -24(%rsp)
    movq    -24(%rsp), %xmm0
    ret
    .p2align 4,,10
    .p2align 3
.L9:
    movss   .LC1(%rip), %xmm3
    movss   (%rdi), %xmm0
    movss   4(%rdi), %xmm2
    xorps   %xmm3, %xmm0
    xorps   %xmm3, %xmm2
    xorps   %xmm3, %xmm1
    jmp .L4
```

Listing 7.3: Generated assembly language code of Listing 7.1

```
subq    $8, %rsp
movaps  (%rsi), %xmm0
movaps  16(%rsi), %xmm1
movaps  32(%rsi), %xmm2
movaps  %xmm2, %xmm3
cmpltps .LC2, %xmm3
movaps  %xmm0, %xmm4
xorps   .LCSPS, %xmm4
movaps  %xmm1, %xmm5
xorps   .LCSPS, %xmm5
movaps  %xmm2, %xmm6
xorps   .LCSPS, %xmm6
mulps   %xmm0, %xmm0
mulps   %xmm1, %xmm1
mulps   %xmm2, %xmm2
andps   %xmm3, %xmm6
andps   %xmm3, %xmm5
andps   %xmm3, %xmm4
movaps  %xmm3, %xmm7
andnps  %xmm2, %xmm7
movaps  %xmm3, %xmm2
andnps  %xmm1, %xmm2
andnps  %xmm0, %xmm3
orps    %xmm7, %xmm6
orps    %xmm2, %xmm5
orps    %xmm3, %xmm4
movaps  %xmm4, (%rdi)
movaps  %xmm5, 16(%rdi)
movaps  %xmm6, 32(%rdi)
addq    $8, %rsp
ret
```

Listing 7.4: Generated assembly language code of Listing 7.2

does this function invoke other functions.[6]

- The constant `.LCSPS` is loaded three times from memory. A single load into a free register and then using this register instead is faster. The scalar version generated by GCC does this.
- The loop setup which is not listed here may also be further optimized.

Although the Swift Compiler's output is longer and thus looks slower at first glance, *4 packed singles*—indicated by the `ps` postfix for *packed singles* in the mnemonics—are used throughout the procedure whereas the scalar version of GCC's output just uses *singles*—indicated by the `ss` postfix for *scalar singles*.[7] Furthermore we can see how the need for branching is eliminated and the masking sequence generated by Algorithm 6.13 is used to select the correct results.

Now the running times of the C++ and the corresponding Swift programs are compared. This also means that work, which must be done beside the actual computation like startup and finishing code or allocation of memory, is also taken into account for both versions. Because of this effect the observed speedups of the calculations are even slightly faster.

The test machine was an *Intel® Core™ 2 Duo T6400* with 2 000 MHz and 3 GiB RAM running the 64 bit x64 version of *Kubuntu* 10.4 (*Lucid Lynx*)[8] Alpha 3 with GCC version 4.4.3. Each benchmark was run 20 times. The average speedups of the Swift implementation are summed up in Tables 7.1 and 7.2. Benchmarks on other machines with enough memory for the huge arrays/SIMD containers yielded comparable results.

7.4 Summary

Although the output of the Swift Compiler Framework still leaves room for further optimizations as mentioned above it outperforms traditional arrays by far. Theoretically, a speedup of the given SIMD length can be expected. The observed speedup roughly acknowledges this hypothesis. The use of *doubles* does not quite achieve this expectation but the observed performance gain is still notable. *Bytes* even exceed this expectation to an amazing average speedup of about 18–22. The most used types in 3-D graphics are perhaps *singles* which were intensively tested in the benchmark. The achieved speedups of about 3–5 are tremendous.

Interestingly, the **ifelse** benchmark with *singles* performs quite well although the if- as well as the else-case must always be executed in contrast to the scalar version. The

[6] Such functions are also called *leaf functions*.

[7] The xorps instruction in Listing 7.3 is used because a xorss instruction does not exist. However, only the lower *single* in the XMM register is needed. The other three *singles* are just garbage and are neither used beforehand nor needed afterwards.

[8] See http://www.kubuntu.org.

	singles	*doubles*	*bytes*	*words*	*doublewords*	*quadwords*
vecadd	5.17	1.56	22.00	9.16	3.96	1.43
vecross	4.71	1.62	n/a	n/a	n/a	n/a
matmul	3.47	1.68	n/a	n/a	n/a	n/a
ifelse	4.92	1.30	n/a	n/a	n/a	n/a

Table 7.1: Average speedup of Swift vs. `g++ -O2`

	singles	*doubles*	*bytes*	*words*	*doublewords*	*quadwords*
vecadd	5.04	1.44	18.16	9.25	4.43	1.37
vecross	4.75	1.73	n/a	n/a	n/a	n/a
matmul	3.13	1.53	n/a	n/a	n/a	n/a
ifelse	4.80	1.23	n/a	n/a	n/a	n/a

Table 7.2: Average speedup of Swift vs. `g++ -O3`

reason for this is that branching is more expensive than executing a couple of instructions in a linear program.

As already mentioned in Section 3.1 the AVX will provide 256 bit wide registers. Thus *8 packed singles* instead of *4 packed singles* can be used, for instance. An additional speed up of two can be expected with this technology. Some GPUs support calculations with half precision floating point types. These floating types—often called *halves*—just use two bytes of memory. While these types are not useful for high precision arithmetic the resolution is sufficient in applications where precision is not that important. Using *halves* instead of *singles* can double the performance again as long as the target hardware supports such calculations.

8 Outlook

In this chapter some more advanced techniques are discussed which can be used to get rid of some restrictions of the algorithms presented in Chapter 6. Furthermore, we have discussed only typical for-each calculations so far. If more sophisticated computations are needed some more work must be done. The concepts presented in this chapter are not perfectly polished solutions. More research in this area is necessary in order to extend the basics of Chapter 6 solidly.

8.1 Getting Rid of Type Restrictions

First of all, some approaches are presented which try to circumvent some constraints related to the types which do not allow vectorization.

8.1.1 Vectorization of Packed Types

Until now the vectorization of *packed types* was not allowed since LengthOf$(t) = -1$ for all $t \in T_{\text{packed}}$. It is not difficult to allow the vectorization. If LengthOf(t) yields $simd_{width}/\,\text{size-of}(t)$ like for *integers* and *floats* the algorithms only need to be adapted slightly. The only prerequisite is that $\text{size-of}(t) \leq simd_{width}$.

8.1.2 SIMD Containers with Known Constant Size

Like mentioned in Section 2.4 there are two flavors of arrays. Similarly, two flavors of SIMD containers are possible: Those already presented in Section 6.1.1 and a variant which only consists of chunk of memory on the stack with an at compile time known size. This variant does not introduce any problems. It rather allows some optimizations since the size is known at compile time.

8.1.3 Allowing Vectorizations with Different SIMD Lengths

In Chapter 6 vectorization is not possible if types with different SIMD lengths are involved. But if simply an SIMD length of $simd_{width}$ is always used the length is obviously the same in all cases while the available SIMD registers are occupied fully. However, this has some drawbacks especially when large types are used. Consider $simd_{width} = 16$ (as implied by SSE) and the use of *doubles*:

- All calculations must be broken down into eight sub-calculations and the resulting assembly code up gets bloated.
- The more sub-calculations are needed the more increases register pressure which in return increases the likelihood for spilling.[1]
- One SIMD block of an SIMD container becomes unnecessarily large. This increases the likelihood for cache misses and the need for paging again.

As a compromise an SIMD length could be chosen which is just enough for the needed data types. Only corner cases where—let us say *bytes* and *doubles* are used simultaneously—yield unfavorable conditions while mixing *singles* and *doubles*, for instance, does not result in a large impact. Hence such a record would become vectorizable and an SIMD statement using *singles* and *doubles* is not a problem anymore.

Another problem is a *record* which only uses *singles*. It would get an SIMD length of $simd_{width}/4$. A statement which uses this *record* and makes use of *doubles*, too, is not vectorizable. One option to deal with this problem is to let the programmer decide, which data types are the largest ones he wants to use. So he can define a *record* only consisting of *singles* but tell the compiler that he wants to use *doubles*, too. Thus an SIMD length of $simd_{width}/8$ would be chosen instead of $simd_{width}/4$.

Either way SIMD statements will not work if *records* with different SIMD lengths are used.

8.1.4 Allowing Pointers

When *pointers* should be allowable it is obvious that the used SIMD lengths must be at least $simd_{width}/\,\text{size-of}(pointer)$. Furthermore, it heavily depends on the approach discussed in the last section: When dereferencing a data structure with an SIMD length which is not equal to the current function's one vectorization is not possible. If, however, the SIMD lengths of the current function's context and the dereferenced data structure match vectorization is not a problem anymore.

8.2 More Complex SIMD Computations

As already said only typical for-each constructs have been discussed until now. Other frequently used operations on containers are possible, too.

[1] *Register pressure* is the number of variables which are live at a point in a program. If this number exceeds the number of available registers a register must be saved to memory while this value must be reloaded on the next use of the corresponding variable. This process is called *spilling*.

8.2.1 Shifting/Rotating of SIMD Containers

If all elements within a container are moved to the right while the first now free element is filled with a standard value we speak of a right shift. Moving all elements to the left is a left shift. If the element which is shifted out on one end is shifted in again at the other end, we speak of right or left rotating, respectively. It is also possible to shift/rotate by more than only one element.

If n elements are involved while $simd_{length} \mid n$, the operation is easily implemented as only complete SIMD blocks must be moved around. The blocks by themselves need not be touched internally. On the other hand if $simd_{length} \nmid n$, each block must be shifted by $s \coloneqq n \bmod simd_{length}$ or $simd_{length} - s$ depending on the direction. The elements, which are shifted out, must be shifted into the next (or previous) block. In the case of rotation the last (or first) elements, which are moved out of the container, must be inserted into the free slots of the first (or last) block.

8.2.2 SIMD Statements with Shifted Loop Indices

A problem arises when elements of SIMD containers should be combined while using different indices. If these indices only differ by an at compile time known constant, vectorization may be done nevertheless:

for $i = 0$ to s **do**
    $j \leftarrow i \pm d$
    $a[i] \leftarrow b[j]$
end for

As long as the absolute difference $d \coloneqq |i - j|$ a multiple of $simd_{length}$ only different SIMD blocks must be used while the internal structure of the blocks need not be changed. If, however, d is not a multiple of $simd_{length}$, one container must be shifted by $s \coloneqq d \bmod simd_{length}$ or by $simd_{length} - s$ depending on the direction.

If d is an at compile time unknown constant, both loop versions can be generated and a runtime check can test which version must be taken.

8.2.3 Reduction

An operation where a given map $op : T \times T \to T$ is recursively applied on all neighbored elements of a container of Ts until a sole element remains, is usually called *reduction*. A classic example is to sum up all elements in a container. As long as the operation is commutative an SIMD approach is simple: Let us assume that a reduction with an SIMD container with the vector addition defined in Listing 6.1 should be performed. A repeated invocation of the SIMD version of **operator** +, while every time two SIMD blocks are reduced to one, yields a single block in the end. This block can be broken down into $simd_{length}$ scalar elements.

8.3 Saturation Integers

SSE provides integer operations with saturation. Normally, an addition or subtraction with integers which causes an overflow is wrapped around. This means, for instance, that an unsigned addition of type t is actually an operation

$$(a + b) \bmod \mathit{width}, \text{ with } \mathit{width} = 2^{\text{size-of}(t)} .$$

This behavior is often the desired effect. But especially for image and audio processing a saturation of an addition or subtraction could be more reasonable:

$$\min(a + b,\, 2^{\text{size-of}(t)} - 1) .$$

We suggest to give a programming language special saturation types which make use of these calculations. Even if the target architecture does not directly support these calculations it is nevertheless beneficial to use such types:

First of all, the source code becomes a bit cleaner if this feature is supported because many if-else-cases or equivalent constructs are eliminated. Second, even if an appropriate instruction is not available for the target ISA, the calculation can be emulated with conditional moves, for instance. This eliminates branches on assembly level, too, which can be quite advantageous for the performance.

8.4 Cache Prefetching

During benchmarking it has been observed that memory accesses are the bottleneck of the execution. Running the same benchmarks as in the last chapter but accessing only the same element in all iterations instead of traversing over the SIMD containers results in a performance increase of roughly one order of magnitude although technically the same number of instructions are executed in both cases.

The x86/x64 processor family features some instructions which can hide cache latencies or prevent cache pollution.[2] Unfortunately, good use of such instructions afford much knowledge of the underlying cache hierarchy. Improperly use of such instructions can easily decrease the performance.

Some experiments with manual adjusted assembly code by the author with prefetch instructions yielded mixed results: Some runs resulted in a notable performance gain others—although using the same assembly code—were slower.

[2] Usually, memory values which are accessed during execution are loaded into the cache optimistically assuming that these values are needed again a short time afterwards. However, if this is not the case other values get thrown out of the cache in order to make room for a value which will not be needed anyway. This effect is called *cache pollution.*

8.5 GPU computing

The techniques and algorithms presented in this thesis can be applied to GPU computing as well. The R5xx chipset by the AMD Graphics Product Group[3] provides a vertex shader which has many instructions for use with *4 packed singles* [cf. AMD, e]. Using the GPU for mathematics with five dimensional vectors, for example, is currently a real performance problem. The paradigm presented in this thesis could be a simple solution.

8.6 Limitations

Algorithms with complicated control flows like parsing can hardly benefit from SIMD programming. Other problems are algorithms, which need random access to the elements in a container, although some constellations are vectorizable nevertheless (see Sections 8.2.2 and 8.2.3).

Thus stream processing, which can be frequently found in 3-D graphics, physical simulation software, image, audio and video processing, is most suitable for SIMD programming.

[3] The company was formally known as *ATI Technologies Inc.* until the company was acquired by AMD in 2006. The brand *ATI* was retained for graphics cards, however.

9 Conclusion

In this thesis we present a new SIMD programming technique. Any programming language with support for arrays and records can be enhanced with this technique. Further research topics and limitations are discussed in Chapter 8.

This SIMD extension does not demand much of a programmer: Merely, the types and functions in questions must be marked with the **simd** keyword and special SIMD statements must be used instead of traditional loops. These statements are intuitive in use and are even less verbose than traditional C-like for-loops. This makes SIMD programming for the first time *modular*, *portable* and *high-level* orientated while maintaining great *performance*.

Especially, the automatic vectorization of functions is very convenient for the programmer: Typical used programming constructs are automatically transformed without involvement of the programmer. More complex control flow structures, however, still need further research although some guidelines are given in this thesis anyway. The transformations can be done in linear time and hence do not increase the asymptotic running time of the compiler.

The benchmarks—which are discussed in Section 7.4 in detail—show that SIMD containers outperform traditional arrays of *records* by a factor, which is roughly equal to the involved SIMD length, because the SIMD registers of the target machine are used with full capacity. This speedup must be regarded as significant—especially for multimedia projects where often large streams of data must be handled.

Bibliography

All URLs referenced in this thesis—including those used in footnotes—have been checked for validity on December 16, 2011.

AMD. *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*. Advanced Micro Devices, Inc., September 2007a. URL http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24592.pdf. Publication No. 24592, Revision 3.14.

AMD. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*. Advanced Micro Devices, Inc., September 2007b. URL http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24594.pdf. Publication No. 24594, Revision 3.14.

AMD. *AMD64 Architecture Programmer's Manual Volume 4: 128-Bit Media Instructions*. Advanced Micro Devices, Inc., September 2007c. URL http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/26568.pdf. Publication No. 26568, Revision 3.10.

AMD. *AMD64 Architecture Programmer's Manual Volume 5: 64-Bit Media and x87 Floating-Point Instructions*. Advanced Micro Devices, Inc., September 2007d. URL http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/26569.pdf. Publication No. 26569, Revision 3.09.

AMD. *Radeon R5xx Acceleration*. Advanced Micro Devices, Inc., October 2009e. URL http://http://www.x.org/docs/AMD/R5xx_Acceleration_v1.4.pdf. Revision 1.4.

A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986. ISBN 0-201-10088-6.

A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998. ISBN 0-521-58388-8.

K. Chung. *Phong Equation Using Intel® Streaming SIMD Extensions and Intel® Advanced Vector Extensions*, December 2009. URL http://software.intel.com/file/24862. Version 3.0.

C. Click and K. D. Cooper. Combining Analyses, Combining Optimizations. *ACM Transactions on Programming Languages and Systems*, 17, 1995.

K. D. Cooper, T. J. Harvey, and K. Kennedy. A Simple, Fast Dominance Algorithm, 2001. URL http://www.cs.rice.edu/~keith/EMBED/dom.pdf.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001. ISBN 0262032937.

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/115372.115320.

E. W. Dijkstra. Goto statement considered harmful. *Communications of the ACM*, 11 (3):147–148, May 1968.

E. Kunst and J. Quade. Kern-Technik — Folge 43. *Linux Magazin*, 1/09:90–93, 2009. URL http://www.linux-magazin.de/Heft-Abo/Ausgaben/2009/01/Kern-Technik.

M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, September 1972.

S. Hack. *Register Allocation for Programs in SSA Form*. PhD thesis, Universität Karlsruhe, October 2007. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/6532.

M. S. Hecht and J. D. Ullman. Flowgraph reducibility. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 238–250, New York, NY, USA, 1972. ACM. doi: http://doi.acm.org/10.1145/800152.804919.

Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, November 2009a. URL http://www.intel.com/Assets/PDF/manual/248966.pdf. Order Number 248966-020.

Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*. Intel Corporation, December 2009b. URL http://www.intel.com/Assets/PDF/manual/253665.pdf. Order Number 253665-033US.

Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M*. Intel Corporation, December 2009c. URL http://www.intel.com/Assets/PDF/manual/253666.pdf. Order Number 253666-033US.

Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z*. Intel Corporation, December 2009d. URL http://www.intel.com/Assets/PDF/manual/253667.pdf. Order Number 253667-033US.

D. E. Knuth. Structured programming with go to statements. *Classics in software engineering*, pages 257–321, 1979.

T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/357062.357071.

M. Matz, J. Hubička, A. Jaeger, and M. Mitchell. *System V Application Binary Interface — AMD64 Architecture Processor Supplement*, January 2010. URL http://www.x86-64.org/documentation/abi.pdf. Draft Version 0.99.4.

B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000. ISBN 0201700735.

Wikipedia. Pure function — Wikipedia, The Free Encyclopedia, 2010. URL http://en.wikipedia.org/w/index.php?title=Pure_function&oldid=339514088. [Online; accessed 27-February-2010].

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, sowie alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen worden sind, als solche kenntlich gemacht habe.

Münster, 16. März 2010 _____

Roland Leißa