Master Thesis

Explicit SIMD instructions into JVM using LMS

Author(s):
Toskov, Ivaylo

Publication Date:
2017-06-25

Permanent Link:
https://doi.org/10.3929/ethz-b-000166022

Rights / License:
In Copyright - Non-Commercial Use Permitted
Abstract

With the ever-growing need for fast and high performance applications, developing software that possesses these properties requires taking the full advantage of the hardware specifics on the target architecture. On the other hand, developing large and complex software systems requires abstraction and generalization. Lightweight modular staging (LMS) fills this gap by providing a framework for building domain-specific languages (DSLs) with zero-cost abstraction. While the modular approach in LMS provides multiple backends for code generation, it lacks a complete support to address the specific instruction sets on different CPUs.

In this work, we extend LMS by introducing single instruction, multiple data (SIMD) instructions to the framework. With the presence of multiple instruction set architectures (ISAs), each containing hundreds or thousands of instructions, implementing them by hand would be cumbersome and error-prone process. Instead, we propose a systematic approach to automatically generate the support for SIMD instructions by given specification.

Our approach addresses the modular aspect of LMS, implementing each instruction as part of an ISA-specific extensible DSL. Furthermore, we aim at providing an intuitive interface to access the SIMD instructions, and demonstrate their usage and extensibility by building another layer of abstraction that consists of a subset of Basic Linear Algebra Subprograms (BLAS). Our implementation is often orders of magnitudes faster than its corresponding Java Virtual Machine (JVM) equivalent, and is directly accessible in the ecosystem of other LMS-based DSLs.
Acknowledgements

I would like to thank Prof. Markus Püschel for his invaluable suggestions throughout the whole project and for being a source of knowledge and inspiration. His impeccable lectures encouraged me to explore an area that was unknown to me before, which greatly broadened my horizon as a computer scientist. Teachers like him are extremely hard to find.

I would also like to thank my direct supervisor, Alen Stojanov, for his remarkable patience, enthusiasm, and knowledge. Despite his busy schedule, he would always devote enough time to carefully answer all my questions and give me motivation when I felt stuck. Not only were his insights tremendously helpful to me, but he would also give me freedom to express my own ideas. His outgoing personality and great sense of humor made the whole experience extremely enjoyable. I wholeheartedly appreciate everything he has done for me.

Finally, I want to express my gratitude to my parents, Petar and Maya, for providing me with continuous support and encouragement throughout the process of researching and writing this thesis. This accomplishment would not have been possible without them.
# Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Objectives and Contributions</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Thesis outline</td>
<td>3</td>
</tr>
<tr>
<td>2 Related work</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Vectorization through code generation</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Explicit vectorization in the JVM</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Automatic vectorization</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Link to native code</td>
<td>6</td>
</tr>
<tr>
<td>3 Lightweight modular staging</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>7</td>
</tr>
<tr>
<td>3.2 Internal representation</td>
<td>8</td>
</tr>
<tr>
<td>3.3 Code generation</td>
<td>10</td>
</tr>
<tr>
<td>3.4 Side effects</td>
<td>12</td>
</tr>
<tr>
<td>3.5 Transformers</td>
<td>15</td>
</tr>
<tr>
<td>4 LMS intrinsics generation</td>
<td>17</td>
</tr>
<tr>
<td>4.1 Base classes</td>
<td>17</td>
</tr>
<tr>
<td>4.2 Generation process</td>
<td>19</td>
</tr>
<tr>
<td>4.3 Type mappings</td>
<td>22</td>
</tr>
<tr>
<td>4.4 Dealing with large size</td>
<td>23</td>
</tr>
<tr>
<td>4.5 Side effects</td>
<td>23</td>
</tr>
<tr>
<td>4.6 Pointer abstraction</td>
<td>24</td>
</tr>
<tr>
<td>4.7 Maven artefact</td>
<td>26</td>
</tr>
<tr>
<td>5 BLAS-like interface</td>
<td>27</td>
</tr>
<tr>
<td>5.1 Pointwise operations</td>
<td>27</td>
</tr>
</tbody>
</table>
CONTENTS

5.2 Dot product .................................................. 30
5.3 Matrix-vector product ................................. 32
5.4 Matrix-matrix product ............................... 33

6 Evaluation .................................................. 35
   6.1 Benchmarking configuration ....................... 35
   6.2 Daxpy .................................................. 36
   6.3 Matrix-matrix multiplication ...................... 37

7 Future work .............................................. 39

8 Conclusion ............................................... 41

Bibliography ............................................. 43
Chapter 1

Introduction

1.1 Overview

There is a large set of applications that rely on high performance – from consumer computing through embedded systems to scientific computing and research. Due to the steady exponential growth of power of computer chips in the last couple of decades, various impactful technologies were introduced to the world. This was possible, since the number of transistors that could fit onto chips doubled approximately every two years – an observation known as Moore’s Law. However, this trend is no longer valid due to physical limitations of the chips. Consequently, other paradigms such as parallel computing emerged in order to address the need for performance. One example technique for exploiting data parallelism is vectorization. Similar to most optimizations, it requires specific knowledge regarding the underlying hardware.

While performance requires concretization, building large and maintainable software systems calls for abstraction and generalization. Consequently, there are no existing solutions that can successfully combine the two. Most performance-critical software is written in low-level languages such as C, whereas programming languages that leverage high-level abstractions are the preferred choice for applications where high productivity is desired. Generative programming is currently an appealing approach for achieving high performance while keeping a high level of abstraction. It accomplishes that by building abstractions on top of composable pieces of low-level code and embedding domain-specific optimizations in the generation process. Lightweight modular staging (LMS) [39] is a runtime code generation approach that was developed following this paradigm. It constitutes a framework for building embedded compilers for domain-specific languages (DSLs). Systems such as Spiral [38] and Delite [11] rely heavily on LMS to achieve high performance.
One shortcoming of LMS is the fact that it does not contain complete set of intrinsic functions for vectorization. Vectorization relies on SIMD instructions that most current CPUs have. The first use of SIMD instructions dates back to the early 1970s, but they gained higher popularity in the late 1990s with the introduction of Intel’s MultiMedia eXtension (MMX). Extensions such as MMX usually introduce additional registers that have increased size and can fit multiple numbers and provide instructions that can operate on those numbers in parallel. Consequently, the performance may be enhanced by a factor depending on how many numbers can fit in a single register.

One way to implement vectorization is through the use of intrinsic functions, also known as *intrinsics*. Intrinsic functions are functions whose implementation is handled specially by the compiler. Instead of calling a function that is contained in the library, the compiler maps intrinsics directly to a single or multiple machine instructions. Due to the benefits of vectorization, many compilers provide intrinsics that expand to SIMD instructions which are contained in most current ISAs.

### 1.2 Objectives and Contributions

The objectives of this thesis are motivated by the fact that there are no present tools for vectorization in LMS and we use code generation to address this shortcoming. Although this approach is generally applicable for any type of processor and instruction set extension, in this work we focus on all intrinsic functions specified in the Intel Intrinsics Guide website [3]. We will refer to them simply as *intrinsics* unless explicitly mentioned otherwise.

The main contributions of this thesis are:

- It proposes a methodology for generating LMS definitions for intrinsics for SIMD instructions based on a specification. The resulting definitions are well-typed, extensible and can undergo further optimizations. Moreover, we empirically verified that they capture side effects correctly, although there are no theoretical guarantees.

- A Maven [5] artefact containing definitions for all 5724 intrinsics and code generation facilities for producing C code with according intrinsic calls. The instruction set extensions that we focus on are MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2, FMA and AVX-512.

- An implementation of a subset of BLAS [2] routines that demonstrates the abstractions that can be built by utilizing the intrinsics package.

- A performance comparison of generated code that uses the intrinsics with an equivalent Scala implementation that was run in the JVM.
1.3 Thesis outline

In this chapter we presented the motivation behind introducing the intrinsics into JVM and the contributions of this work. The rest of the thesis is organized as follows:

- **Chapter 2** presents work that is related to vectorization in the JVM.
- **Chapter 3** constitutes a short introduction to LMS.
- In **Chapter 4** we describe the intrinsics generation process and the accompanying challenges.
- In **Chapter 5** we illustrate example usages of the intrinsics. More specifically, we focus on subset of BLAS routines.
- **Chapter 6** presents performance evaluation between implementations in Scala and vectorized C versions generated by LMS using the intrinsics.
- **Chapter 7** proposes ideas for future work.
- **Chapter 8** concludes the thesis.
Chapter 2

Related work

Given that we use code generation to introduce intrinsics to LMS, it is important to examine other approaches that rely on the same technique for producing vectorized code. Moreover, this thesis provides an interface for explicit vectorization in the JVM, so we should investigate the work done so far in that area. Furthermore, there have been various strategies for automatic vectorization. Finally, an alternative for achieving performance in the JVM is to link it to native code. In the rest of this chapter we are going to examine the state of the art in all these directions.

2.1 Vectorization through code generation

There have been various approaches using code generation to produce vectorized code. Most of them exploit domain-specific knowledge of a certain problem and generate SIMD instructions accordingly. For example there are proposed solutions for generating vectorized code for digital signal processing [38, 18] and fast Fourier transform in particular [21, 28, 20, 19]. Other domain-specific approaches include the generation of customized code for stencils [24] and restructuring programs for exposing ISA-independent vectorizable codelets [26].

2.2 Explicit vectorization in the JVM

There is not much research regarding explicit vectorization in the JVM. Currently the user has no control over whether the code will be vectorized or not. There is neither an interface for generating SIMD code, nor is it possible to give any hints to the compiler to perform vectorization. However, as of time of writing, there is an ongoing research at Oracle for developing an API that can leverage SIMD instructions on modern CPUs [10]. Their proposal relies on constructs called code snippets that can be interpreted as HotSpot in-
2. Related work

trinsics. The work presented by Nie et al. [32] introduces Java vectorization interface (JVI) that also provides an API for explicit vectorization.

2.3 Automatic vectorization

Most state-of-the-art compilers provide some form of auto-vectorization. The most popular technique is superword level parallelism (SLP) [27]. It works by identifying isomorphic statements within basic blocks. Jikes RVM [16] implements SLP to automatically vectorize code by using a modified tree-pattern matching algorithm to identify similar consecutive instructions and transform them into equivalent vector instructions. Vapor SIMD [33] suggests another design that relies on split-compilation [13] and is able to vectorize not only straight-line code, but inner and outer loops as well.

There are various approaches for automatic vectorization that are not restricted to the JVM, but whose underlying concepts are generally applicable. The strategy proposed by Stock et al. [40] relies on machine learning for predicting the performance of given set of SIMD instructions. Another example is Selftrans [31] which is a system that applies automatic vectorization techniques to binary code at runtime.

2.4 Link to native code

Although various attempts have been made in order to introduce vectorization into the JVM, most developers still resort to native code when performance is critical. For example the most popular linear algebra libraries in Java, netlib-java [7] and jBlas [4], are simple wrappers for BLAS routines. The performance comparison between Java and C code linked to the JVM via Java Native Interface (JNI) in the publication by Nassim et al. [23] indicates that native implementations outperform their Java equivalents for compute-bound problems.

However, JNI is not the only option for linking Java to native code. The Gnu Compiler for Java (GCJ) can translate Java programs to native code by using the GNU Compiler Collection (GCC) to compile both Java source code and bytecode into native code. This approach is particularly neat with regard to automatic vectorization, since many techniques are embedded in GCC [35].
3.1 Introduction

The code generation methodology that we propose in this work requires compiler infrastructure that can transform a piece of code into an internal representation and can handle side effects and rewrites. Thus, our approach can be applied to any framework that provides this functionality. We chose Lightweight modular staging [39] to demonstrate our strategy due to its flexible internal representation and explicit control of effects.

LMS is a dynamic multi-stage code generation approach for implementing domain-specific languages embedded in Scala. It harnesses the type system and does not rely on syntactic annotations. Moreover, it uses component technology that allows programmers to integrate domain-specific optimizations into the generation process.

LMS transforms the code written in a DSL into an intermediate representation (IR). This process is known as staging and goes back to at least Jørring and Scherlis [25]. The advantage of this approach is that certain code fragments are executed less often and at a time when the performance is less critical.

LMS is also lightweight, since it only relies on Scala’s type system. Therefore, in order to lift a language construct into its internal representation only the type for this construct needs to be changed. Furthermore, LMS is simply a Scala library that can easily be imported. Thus, there is no need to implement the components of a typical compiler.

In the rest of this chapter we are going to introduce the basics of LMS that are necessary for understanding the generation of the intrinsics.
3. Lightweight modular staging

3.2 Internal representation

As discussed in the previous section, LMS transforms the code into an internal staged representation and this is achieved by changing the type of the computation. Let us illustrate that with an example of getting the maximum of two integer numbers. Normally one would write a function of the following form:

```scala
def max(a: Int, b: Int): Int = {
  if (a > b) a else b
}
```

The only thing that we need to change in order to turn this function into a staged one are the types in the signature:

```scala
def max(a: Rep[Int], b: Rep[Int]): Rep[Int] = {
  if (a > b) a else b
}
```

A variable of type `Rep[Int]` represents a staged computation that will yield a result of type `Int`. With this simple modification we staged the function `max` and we might perform further optimization on its internal representation.

At this point we do not need to worry how is comparison for variables of type `Rep[Int]` implemented. We only need to know that LMS comes with an implementation of basic operations for staged types and that control structures are implemented using the Scala-virtualized compiler [30].

It is important to understand how is the internal representation defined. LMS contains a trait `Base` that is the basis for all DSLs:

```scala
trait Base {
  type Rep[+T]
}
```

The declaration `type Rep[+T]` defines an abstract type constructor [29] which will determine the representation of the staged expressions. This means that it does not specify a concrete implementation, but rather postulates the existence of one.

Although LMS does not restrict the programmer with the choice of internal representation, it comes with a ready implementation that is based on “sea of nodes” [12]. This representation is a directed graph that can be accessed through a tree-like interface.

Figure 3.1 illustrates a minimized version of the original Expressions trait in LMS. The basic objects of interest are expressions (subclasses of Exp) and
3.2. Internal representation

definitions (subclasses of \texttt{Def}). Expressions are atomic and are either constants (class \texttt{Const}) or symbols (class \texttt{Sym}). Definitions represent composite operations that combine multiple expressions. The intrinsics that we generate are in fact definitions. An example definition that represents the addition of two integers would have the following form:

\begin{verbatim}
  case class IntPlus(lhs: Exp[Int], rhs: Exp[Int]) extends Def[Int]
\end{verbatim}

\texttt{IntPlus} is a definition that will yield a result of type \texttt{Int} and takes two integer expressions as input.

As shown in Figure 3.1, LMS also defines 3 methods that convert symbols to definitions and vice versa (implementation is omitted for brevity). The underlying idea is that definitions are meant to be associated with symbols and only be referenced via their symbols. That means that all composite values will be named similar to administrative normal form (ANF) [17].

\begin{verbatim}
trait Expressions {
  // expressions (atomic)
  abstract class Exp[+T]
  case class Const[T](x: T) extends Exp[T]
  case class Sym[T](n: Int) extends Exp[T]

  // definitions (composite, subclasses provided by other traits)
  abstract class Def[T]

  def findDefinition[T](s: Sym[T]): Option[Def[T]]
  def findDefinition[T](d: Def[T]): Option[Sym[T]]
  def findOrCreateDefinition[T](d: Def[T]): Sym[T]

  // statement (links syms and definitions)
  abstract class Stm

  // typed pair of symbols and definitions
  case class TP[+T](sym: Sym[T], rhs: Def[T]) extends Stm

  // list of all global definitions
  var globalDefs: List[Stm] = Nil

  // bind definitions to symbols automatically
  implicit def toAtom[T](d: Def[T]): Exp[T] =
    findOrCreateDefinition(d)
}
\end{verbatim}

\textit{Figure 3.1:} Expression representation (method implementations omitted).

The way definitions are mapped to symbols is through statements (class \texttt{Stm}). A typed pair (class \texttt{TP}) is a concrete implementation of \texttt{Stm} and is used for most of the DSLs that come with LMS. It simply packs both the symbol and the definition within its constructor.
As we have already mentioned, definitions are only accessed through their corresponding symbols. Thus, the implicit method `toAtom` is used to retrieve the symbol that is associated with a given definition. That way the generated IR is guaranteed to be in Static Single Assignment (SSA) [15] Form. The variable `globalDefs` keeps track of all definitions encountered so far and `toAtom` either returns a symbol for a definition that is already contained in the list or creates a new one and stores it in `globalDefs`. This is how common subexpression elimination is performed automatically.

### 3.3 Code generation

Given that the intrinsics that we introduced are implemented in C, we will investigate how is C code generated from the internal representation that we described in the previous section. LMS defines a trait called `GenericCodegen` that is the base for all code generation implementations. Out of the many methods that it specifies, three are of high importance for our project:

- **emitNode** This method takes a definition and its symbol and produces the corresponding code depending on the actual implementation.
- **remap** This method takes a type of a variable and remaps it to another type depending on the preferences of the DSL designer. This is crucial in our case, since Scala and C have different type systems.
- **isPrimitiveTyp** This method also takes a type as input and returns true in case the type is considered primitive, false otherwise.

We should illustrate the generation process using a naive staged function that computes the interquartile range (IQR) [42] of the elements of a sorted array `a`:

```scala
1 def iqr(a: Rep[Array[Int]], len: Rep[Int]): Rep[Int] = {
2 a((len * 3) / 4) - a(len / 4)
3 }
```

LMS transforms this code into the following internal representation:

```plaintext
1 TP(Sym(2), IntTimes(Sym(1), Const(3)))
2 TP(Sym(3), IntDivide(Sym(2), Const(4)))
3 TP(Sym(4), ArrayApply(Sym(0), Sym(3)))
4 TP(Sym(5), IntDivide(Sym(1), Const(4)))
5 TP(Sym(6), ArrayApply(Sym(0), Sym(5)))
6 TP(Sym(7), IntMinus(Sym(4), Sym(6)))
```

The symbols with identifiers 0 and 1 are the input arguments. Due to the fact that they are expressions, they are not in the list of statements. We do not need to worry about the actual implementation of the definitions that are being produced by LMS.
3.3. Code generation

Now we need a simple generator that will produce C code for this internal representation. Firstly, we need to specify how the types should be remapped:

```scala
override def remap[A](m: Typ[A]) : String = {
  m.toString match {
    case "Int" => "int32_t"
    case _ if m.erasure.isArray => remap(m.typeArguments.head) + "*
    case _ => super.remap(m)
  }
}
```

The parameter \( m \) corresponds to the type that needs to be remapped. We use its `toString` method for brevity, although this is not the recommended way to perform pattern matching on types. We remap the Scala type `Int` to C type `int32_t`. With \( m.erasure.isArray \) we detect that the type in question is an array so we remap its base type with `remap(m.typeArguments.head)` and add an asterisk afterwards to specify that it is a pointer.

With a simple implementation of `isPrimitiveType` we can indicate that our new type `int32_t` is primitive:

```scala
override def isPrimitiveType(tpe: String) : Boolean = {
  tpe match {
    case "int32_t" => true
    case _ => super.isPrimitiveType(tpe)
  }
}
```

The only thing left is to implement generation rules for the definitions in our IR:

```scala
override def emitNode(sym: Sym[Apply], rhs: Def[Apply]) = {
  rhs match {
    case ArrayApply(x, n) => emitValDef(sym, s"$x[$n]"
    case IntDivide(a, b) => emitValDef(sym, quote(a) + " / " + quote(b))
    case IntTimes(a, b) => emitValDef(sym, quote(a) + " * " + quote(b))
    case IntMinus(a, b) => emitValDef(sym, quote(a) + " - " + quote(b))
    case _ => super.emitNode(sym, rhs)
  }
}
```

In this snippet we use helper methods that make the generation process more convenient. The first argument of `emitValDef` is a symbol that will be on the left-hand side of an assignment and the second argument will go to the right. The method `quote` simply returns a variable or a constant for a given expression.

The produced code by our generator would have the following form:
3. Lightweight modular staging

```c
int32_t iqr(int32_t* x0, int32_t x1) {
    int32_t x2 = x1 * 3;
    int32_t x3 = x2 / 4;
    int32_t x4 = x0[x3];
    int32_t x5 = x1 / 4;
    int32_t x6 = x0[x5];
    int32_t x7 = x4 - x6;
    return x7;
}
```

3.4 Side effects

A lot of the intrinsics perform writes and OS-targeted operations. For that reason, it is also important to understand how LMS handles side effects. Before we investigate the approach of LMS to tackle this issue, we should briefly examine how the scheduling of IR nodes works.

As we mentioned earlier in section 3.2, definitions are compositions of expressions. In order to schedule the statements in the correct order, LMS needs to make sure that each definition comes after expressions that it depends on. Thus, a dependency graph is built and an execution order is determined based on its topological sort [14, Chapter 22.4]. When LMS is instructed to create a schedule for a block of expressions, it starts from the final expression, known as the block result, and tracks all of its dependencies. Expressions that cannot be reached from the final expression are not part of the generated code. This is how dead code elimination (DCE) is automatically provided by LMS.

Although the order of execution for some expressions is fixed, this is not the case for expressions that do not depend on each other. Figure 3.2 illustrates the dependency graph of the \texttt{iqr} method that we introduced in Section 3.3:

![Figure 3.2: Dependency graph of the \texttt{iqr} method introduced in 3.3](image)
3.4. Side effects

While it is obvious that Sym(3) should be scheduled after Sym(2), it is not clear whether Sym(2) should be positioned before Sym(5) or vice versa. The expressions in this example are pure, so the actual order of execution is flexible and LMS can perform code motion. However, this is not the case for expressions that have side effects.

In order to illustrate the influence of effects on scheduling, we should examine the following example for computing the square root of \( x \):

```scala
def sqrt(x: Rep[Double]): Rep[Double] = {
  print("Start of sqrt")
  val y = if (x < 0.0) {
    print("Value of x is negative")
    Const(-1.0)
  } else {
    math_sqrt(x)
  }
  print("End of sqrt")
  y
}
```

Firstly, we should understand what is happening within the first block of the if-expression. If the parameter \( x \) has a negative value, \( y \) would be set to \(-1.0\). However, we should note that there is a print statement inside as well and the final expression of the block does not have a direct dependency on it. Thus, LMS needs to keep track of side effects separately.

LMS uses two special nodes to track effect dependencies called \texttt{Reflect} and \texttt{Reify}. The reflect node wraps an effectful definition and adds information on its effect dependencies and the type of effect. The reify node wraps a block result expression, summarizes all the effects and keeps track of the effect dependencies of the whole block. Consequently, effectful statements are tied to their enclosing block and cannot be moved outside. This is the intended behavior, since we want to print to the standard output only if the body of the block is reached.

To further explain the details regarding \texttt{Reflect} and \texttt{Reify} we should examine the internal representation of the \texttt{sqrt} function:
Each \texttt{Reflect} node contains the following three arguments:

- The first argument is the wrapped definition that has side effects.
- The second argument is a summary that describes the type of effect of the node. It might be a write or a read to another symbol, an I/O operation or a global side effect.
- The third argument is a list of all the effect dependencies of the enclosed definition.

In our running example there are four definitions that are enclosed within a \texttt{Reflect} node. The first effectful statement prints the message "Start of sqrt". It is an I/O effect (the arguments of the summary are omitted for brevity) and has no effect dependencies, since it is the first statement that appears in the block. The third symbol relates to the print statement inside the if block and has the very same structure, but also depends on the first print statement. Given that the \texttt{IfThenElse} node contains effectful blocks, it is also wrapped in a \texttt{Reflect} node and it inherits all the outward dependencies of the statements in those blocks. The last print statement (symbol 7) is another I/O effect that depends on the if-then-else statement only and should not directly depend on the print statements before it.

A \texttt{Reify} also consists of three parts, but they play a slightly different role:

- The first argument contains the final expression of a block.
- The second argument contains a combined summary of all the effects within that block.
- The third arguments is a list of all effectful expressions inside the block.

There are two reify nodes, since that is the number of blocks that contain effectful statements. The first one is the first block of the if-then-else expression. The final expression is the constant \texttt{-1.0}, the summary is the same as the one of its only print statement and the symbol of this print statement is the only element of the list of \texttt{Reflect} nodes within that block. The second \texttt{Reify} node denotes the return value of the whole function, which in that example is determined by the return value of the whole \texttt{IfThenElse} node. The summary of this reify node is a combination of the summaries of all three effectful statements within the block. The symbols of those statements are listed in the third argument of the \texttt{Reify} node. As shown above, it contains the first and the last print statements and the if-then-else expression, but not the symbol of the print statement inside the first if branch, since its effect is taken into consideration in the summary of the \texttt{IfThenElse} node.
3.5 Transformers

In the previous sections we briefly mentioned how LMS can perform some optimizations such as code motion, CSE and DCE automatically. However, we might want to explicitly perform some more complex domain-specific optimizations. This is the reason why transformers were introduced to LMS.

Transformers simply go through the schedule and change or replace existing nodes, depending on the optimization that should be performed. If a node does not get transformed, it should make sure that its dependencies are updated in case any of them was transformed. This process is called mirroring. There are many involved optimizations that can be performed on the intrinsics definitions, so it is important to understand the challenges that come with transformations and mirroring.

There are many transformers that focus on the current and past statements to determine the type of optimization that will be performed. These transformers are called forward transformers, since they follow the schedule of the nodes from beginning to end and apply the desired transformations consecutively. Those transformers are also usually substitution transformers, which means that they contain a mapping from old to transformed expressions. Thus, whenever we invoke mirroring on a node whose dependencies have changed, the node will be re-created with updated dependencies that will be retrieved from this substitution mapping.

After the transformer has traversed all the statements in the IR, the scheduler is being called again. This means that code motion, CSE and DCE will be performed on the transformed IR as well.

To illustrate the way statements are transformed, let us examine the following example function which performs some trivial arithmetic operations on the input value $x$:

```scala
def arith(x: Rep[Int]): Rep[Int] = {
  val y = x * 1
  y * 2
}
```

This staged function will result in the following C code:

```c
int arith(int x0) {
  int x1 = x0 * 1;
  int x2 = x1 * 2;
  return x2;
}
```

Firstly, we notice that the input gets multiplied with 1, which is an unnecessary operation, so we want to avoid computing it. Moreover, we would like to transform the multiplication with 2 to a left shift. In order to accom-
3. **Lightweight modular staging**

To accomplish that, we could write a class that extends `ForwardTransformer` which overrides its `transformStm` method:

```scala
override def transformStm (stm: Stm) = stm match {
  case TP(_, IntTimes(sym, Const(1))) => apply(sym)
  case TP(_, IntTimes(sym, Const(2))) => IntShiftLeft(apply(sym), Const(1))
  case _ => super.transformStm(stm)
}
```

The method `transformStm` will be invoked for each statement and we can detect nodes with certain structure through pattern matching. In this example we transform every `IntTimes` that multiplies an integer expression with one by the expression itself. Moreover, once a symbol is multiplied with 2, we replace it with a shift of one position to the left. After we apply the transformer on the function `arith` we get the following transformed result:

```scala
int arith(int x0) {
  int x3 = x0 << 1;
  return x3;
}
```

Note that we invoke the `apply` method of the transformer on all expressions that are needed for the transformed node. Due to the fact that we update the first expression `x1` with `x0`, it is important that we take this substitution into account when we perform the second transformation. This is why it is crucial to implement the mirroring of the intrinsics definitions correctly in order to be able to perform domain-specific optimizations with transformers on top of them.
In this chapter we are going to investigate the intrinsics generation process. There are two main files that are of utmost importance for our project: IntrinsicsBase and IntrinsicsGenerator. The trait IntrinsicsBase contains all the information needed to specify the structure of a given intrinsic. This includes the introduction of new types, list of possible categories, and details regarding headers and performance. The class IntrinsicsGenerator is responsible for the actual generation of all intrinsics based on a given specification. The specification that we use in this work is an XML file that can be found on the Intel Intrinsics Guide website [3]. However, our approach could be applied to a specification in any given format. The result of the generation is a set of traits that contain the Scala implementation of the intrinsics.

The following sections will take a closer look at the generation process, the design of the intrinsics classes and the way the different challenges were handled.

4.1 Base classes

As already mentioned, IntrinsicsBase is the trait that constitutes the basis for the intrinsics. It is extended by traits that correspond to different instruction set architectures (ISAs). It contains the following information:

- Different abstract classes that correspond to intrinsics specific types. Those will be described in Section 4.3.

- Implicit functions that supply the runtime type information of the types introduced in IntrinsicsBase. LMS has an abstract class called Typ that contains this information and can be seen as an equivalent of Scala’s manifests.
4. LMS intrinsics generation

- An enumeration called `IntrinsicsType` that corresponds to the type of the intrinsic. It has three possible values: `FloatingPoint` denoting floating point operations, `Integer` denoting operations on integral numbers and `Mask` denoting operations based on bit masks.

- An enumeration called `IntrinsicsCategory` that maps to a category of the intrinsic. For example, an intrinsic might perform an arithmetic operation, a load or an OS-targeted call. There are 24 categories in total. The full list can be found on the Intel Intrinsics Guide website [3].

- An abstract class called `IntrinsicsDef` that extends the class `Def` of LMS and corresponds to a single intrinsic. It has a generic type that determines the return type of the intrinsic. The contents of this abstract class will be introduced in Section 4.2.

- An abstract class called `Container` that specifies the way common operations are performed on pointers. More information can be found in Section 4.6.

- A method called `isIntrinsicType` that returns `true` for all the types introduced by `IntrinsicsBase`.

- An overridden method `isPrimitiveType` that serves the same purpose as the method with the same name defined in `IntrinsicsBase`.

Moreover, there is an accompanying generator class called `CGenIntrinsics`. It extends the `CCodegen` that is part of LMS and is inherited by the corresponding generators for the different ISAs. It contains the following information:

- A field `headers` that is a set containing all the specific headers that are needed for the proper generation of the intrinsics that are part of the corresponding internal representation. The field is accessed through a getter method called `getIntrinsicsHeaders`.

- An overridden method `remap` that takes care of the proper remapping of intrinsics specific types. More information can be found in Section 4.3.

- An overridden method `isPrimitiveType` that serves the same purpose as the method with the same name defined in `IntrinsicsBase`.

- A method called `resetIntrinsicsHeaders` that empties the set `headers`.

More intuition on how those traits are used is introduced in the following sections.
4.2 Generation process

As we have already mentioned, the generation of the intrinsics is based on a specification that is contained in an XML file that could be found in the Intel Intrinsics Guide website [3]. The file is called data-3.3.16.xml and contains a list of all the Intel SIMD intrinsics. The number 3.3.16 indicates the version of the file and is being updated every time any changes are introduced.

The definition of a single intrinsic is enclosed in an XML element called intrinsic which contains all the relevant information. This information is translated to a Scala case class [36, Chapter 15] that inherits from the Def class in LMS. A single intrinsic contains the following elements:

```
<intrinsic tech="SSE" rettype="__m128" name="_mm_add_ps">
  <type>Floating Point</type>
  <CPUID>SSE</CPUID>
  <category>Arithmetic</category>
  <parameter varname="a" type="__m128" />
  <parameter varname="b" type="__m128" />
  <description>
    Add packed single-precision (32-bit) floating-point elements in "a" and "b", and store the results in "dst".
  </description>
  <operation>
    FOR j := 0 to 3
      i := j*32
      dst[i+31:i] := a[i+31:i] + b[i+31:i]
    ENDFOR
  </operation>
  <instruction name="addps" form="xmm, xmm"/>
  <header>xmmintin.h</header>
</intrinsic>
```

Each element serves a specific purpose:

- **intrinsic** As already mentioned, this is the main element that contains the whole information for a single intrinsic. It corresponds to an LMS definition and is associated with a case class and a function that creates an instance of this case class. It also contains the following attributes:
  
  - **tech** This attribute corresponds to the instruction set extension that introduced the specific intrinsic. We used the value of this attribute for the name of the Scala trait that will contain the intrinsic. In the running example the intrinsic will be placed within a trait called SSE that will extend IntrinsicsBase. Moreover, a trait called CGenSSE will be generated that extends CGenIntrinsics and will contain the rules that determine the generation of all intrinsics which are part of SSE.

  - **rettype** This attribute determines the return type of the intrinsic.
Thus, the case class of that intrinsic will extend `IntrinsicsDef` using this type to replace the generic type parameter. In the running example the case class will extend `IntrinsicsDef[_m128].`

- `name` This attribute corresponds to the original name of the intrinsic. A method with the same name will be created that will return the case class corresponding to this intrinsic. The case class itself will also be placed in the trait of the corresponding ISA and will have the same name as the method, but written in capital letters and with removed underscores in the beginning.

- `type` This element defines the type of the intrinsic. It corresponds to the enumeration `IntrinsicsType` introduced in the previous section.

- `CPUID` The `CPUID` element is similar to the `tech` attribute in `intrinsic`, but it is more specific. For example, there are multiple AVX-512 extensions such as `AVX-512VL` and `AVX-512F` and an intrinsic might be supported by one or more of them. In that case the `tech` attribute will contain the superset of extensions, e.g. AVX-512, whereas the exact list of extensions will be listed using one or multiple `CPUID` elements.

- `Category` This element specifies the category or the list of categories the intrinsic belongs to. It corresponds to the enumeration `IntrinsicsCategory` introduced in the previous section.

- `parameter` Each intrinsic contains zero, single or multiple parameter elements. Those refer to the input parameters of the intrinsic function. Each parameter has two attributes:
  - `varname` The name of the parameter. We reuse the names for the intrinsics definition.
  - `type` The type of the parameter. This type is remapped to a Scala type during the generation process. The remapping is explained in detail in Section 4.3.

- `description` This element contains a short description of the intrinsic. We include this information as a comment for the corresponding intrinsic definition.

- `operation` This element describes the precise operation that is being performed. We ignore this information for the sake of conciseness, since it is not relevant for the internal representation of the corresponding intrinsic.

- `instruction` This element specifies which Assembly instruction is being used for this intrinsic. It has two attributes:
4.2. Generation process

- **name** The name of the Assembly instruction that is being executed by this intrinsic.

- **form** The specific registers that are used by the Assembly instruction.

Again, we ignore this element for sake of conciseness.

- **header** Contains the C header that contains the corresponding intrinsic. We include this information as a field in the corresponding `IntrinsicsDef`.

The abstract class `IntrinsicsDef` that all intrinsics extend has the following structure:

```scala
abstract class IntrinsicsDef[T:Manifest] extends Def[T] {
  val category: List[IntrinsicsCategory.IntrinsicsCategory]
  val intrinsicType: List[IntrinsicsType.IntrinsicsType]
  val performance: Map[MicroArchType, Performance]
  val header: String
}
```

As shown above, it extends the `Def` class defined in LMS. The return type of the intrinsic T is context bound by a Scala `Manifest` which is used to carry runtime type information. Moreover, it contains a list of categories and types that the intrinsics belongs to. Those were already introduced in Section 4.1. The performance field maps an enumeration type corresponding to the microarchitecture to a `Performance` class. The performance class is simply a wrapper for optional double fields for the throughput and latency of the intrinsic. However, in the initial version of the LMS intrinsics library this map is empty, since the performance information was migrated away for newer versions of the XML file in the Intel Intrinsics Guide website [3].

For example, the case class generated from the XML snippet introduced earlier in the section has the following structure:

```scala
/**
 * Add packed single-precision (32-bit) floating-point elements in "a" and "b", and store the results in "dst".
 * a: __m128, b: __m128
 */

case class MM_ADD_PS(a: Exp[__m128], b: Exp[__m128])
  extends IntrinsicsDef[__m128] {
  val category = List(IntrinsicsCategory.Arithmetic)
  val intrinsicType = List(IntrinsicsType.FloatingPoint)
  val performance = Map.empty[MicroArchType, Performance]
  val header = "xmmintrin.h"
}
```

The method that takes care of the creation of an IR node based on this definition has the following implementation:
4. LMS intrinsics generation

```scala
def _mm_add_ps(a: Exp[..m128], b: Exp[..m128]): Exp[..m128] = {
    MM_ADD_PS(a, b)
}
```

As shown above, the signature of the method looks very similar to the original structure of the intrinsic.

Finally, for sake of completeness, we illustrate how does the generator of the ISA containing this intrinsic generate code for it:

```scala
case iDef@MM_ADD_PS(a, b) =>
    headers += iDef.header
    emitValDef(sym, s"_mm_add_ps(${quote(a)}, ${quote(b)})")
```

4.3 Type mappings

One of the challenges that were faced during the generation process was to come up with proper mapping of the types used by the intrinsics. For some of the types this is an easy task, since there is an one-to-one correspondence between C and Scala. The following table illustrates those mappings:

<table>
<thead>
<tr>
<th>C type</th>
<th>char</th>
<th>short</th>
<th>int/size_t</th>
<th>long/long long</th>
<th>float</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala type</td>
<td>Byte</td>
<td>Short</td>
<td>Int</td>
<td>Long</td>
<td>float</td>
<td>Double</td>
</tr>
</tbody>
</table>

We used a Scala library for unsigned numbers called passera [8], since Scala does not provide support for unsigned types. This library introduces types that start with the capital letter `U` followed by the name of the signed equivalent. The mapping of those types looks as follows:

<table>
<thead>
<tr>
<th>C type</th>
<th>unsigned char</th>
<th>unsigned short</th>
<th>unsigned int</th>
<th>unsigned long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala type</td>
<td>UByte</td>
<td>UShort</td>
<td>UInt</td>
<td>ULong</td>
</tr>
</tbody>
</table>

Given that there is no open implementation that contains the intrinsics specific types, they are introduced in IntrinsicsBase as abstract classes. It contains the following types: _.64, _.128, _.128i, _.128d, _.256, _.256i, _.256d, _.512, _.512i and _.512d. Obviously, they have the same names as the corresponding C types. All other masks and enumerations are remapped to Int, since they are typedefs for integers anyways.

The type void was simply remapped to Scala’s Unit. However, void pointers required some special care. The way void pointers are remapped depends on the specific scenario:

- In case a void pointer is an argument of an intrinsic, a generic type `T` is used to replace the void keyword.
4.4 Dealing with large size

- When a void pointer occurs as a return type, it is remapped to an abstract class called `VoidPointer` that is defined in `IntrinsicsBase`. This is required because of the strategy used to handle pointers in general. The details around this strategy will be introduced in Section 4.6.

- When the type is a pointer of a void pointer, it gets remapped to an abstract class `DoubleVoidPointer` that is again contained in `IntrinsicsBase`. The reason behind this decision is the same as for void pointer return types.

The `const` keyword was simply ignored for all types, since Scala arguments are always immutable.

The way pointers were handled is described in Section 4.6.

### 4.4 Dealing with large size

Another challenge that was faced during the generation of the intrinsics was to deal with the large number of intrinsics contained in the data-3.3.16 XML file. The total number is 5724 and 3519 of those are part of AVX-512. Due to the fact that there is a 64KiB limit on the bytecode size of a single method [22], we had to come up with a strategy to split some traits into multiple subtraits, otherwise the `mirror` method gets too large. Thus, we set a limit of maximum number of intrinsics per trait to 175. When a trait contains more than 175 intrinsics, it gets split to multiple traits with added two-digit indexing in the end of their name. For example, the trait `SSE2` distributes all the SSE2 intrinsics to two traits `SSE200` and `SSE201` and simply extends those. Analogous approach is applied to the `CGenSSE2` trait.

### 4.5 Side effects

Given that many intrinsics perform reads and writes or have general side effects, we had to come up with a strategy to take those into account. As a consequence of the fact that we use code generation due to the large number of intrinsics, it is not feasible to take care of the effects of each intrinsic separately. Moreover, there is no explicit information contained in the Intel Intrinsics Guide that identifies which intrinsics are pure and which are not. Thus, we had to make some assumptions in order to tackle this issue. This is our strategy for distinguishing intrinsics with side effects:

- In case an intrinsic returns a pointer, we consider the corresponding definition to be mutable.

- If an intrinsic belongs to the category `load` we mark a read to its definition. There is more information on how we perform reads and writes in Section 4.6.
4. LMS intrinsics generation

- In case an intrinsic has one or multiple pointers as input parameters and is not a load, we reflect a write on all the corresponding definitions. That way we ensure that the order of instructions that involve pointers will be serialized correctly. In the worst case we would perform an unnecessary write, which will make it impossible for LMS to perform code motion. However, the correctness of the generated program is guaranteed.

- In case the return type of an intrinsic is `void`, we perform a simple side effect.

- In all other cases we return a pure definition.

4.6 Pointer abstraction

One problem that we faced is that pointers in C do not have a conceptual equivalent in Scala. Thus, we had to come up with a strategy to somehow model them in Scala. Moreover, given that we do not have a fixed pointer representation, we need to decide on how we are going to track side effects related to pointers.

In order to tackle those issues, two important design decisions were made. Firstly, we use a higher-kinded type for the representation of the actual pointer. That way we give the flexibility to the user of the LMS intrinsics to pick their choice for a class corresponding to a C pointer. Moreover, we introduced an abstract class `Container` that is being passed implicitly to all intrinsics methods that contain pointers. The purpose of this class is to encapsulate all operations that might have side effects. This is needed, since some pointer implementations might potentially require extra considerations that are not taken into account into the general strategy of LMS for handling effects. The `Container` class contains the following methods:

``` scala
abstract class Container[C[_]] {
  def write[A:Typ, T:Typ](c: Exp[C[T]]*)(writeObject: Def[A]): Exp[A]
  def read[A:Typ, T:Typ](c: Exp[C[T]]*)(readObject: Def[A]): Exp[A]
  def apply[T:Typ](c: Exp[C[T]], i: Exp[Int]): Exp[T]
  def update[T:Typ](c: Exp[C[T]], i: Exp[Int], elem: Exp[T]): Exp[Unit]
  def newInstance[T:Typ](size: Exp[Int]): Exp[C[T]]
  def applyTransformer[A](x: Exp[A], f: Transformer): Exp[A]
}
```

As already discussed in the previous paragraph, the higher-kinded type parameter `C` determines the type in Scala that will represent a C pointer. For example a class extending `ContainerArray` will use Scala’s `Array` class. Moreover, the abstract class has the following abstract methods:

- `write` This curried [41] method specifies how to perform write effects. The generic type `T` determines the type of the pointer and the type
4.6. Pointer abstraction

A the type of the definition that performs the write. In most cases the two will be identical. The parameter c contains single or multiple instances of the higher-kinded type C that will be written on. The parameter writeObject is the definition that performs the write. In our case that will be an intrinsic definition.

- **read** This method has a completely identical signature to the write method, but performs a read instead.

- **apply** This method determines how are elements retrieved from a pointer. The parameter c is the instance representing the pointer and the parameter i specifies the offset. The return value will be of type T that depends on the type of the pointer.

- **update** This is the method that specifies how to update an element within the memory region the pointer points to. Again, the parameter c corresponds to the representation of the pointer, i determines the position and elem is the new element that will be placed at that position.

- **newInstance** Given size and type T, this method will determine how to instantiate a pointer for a memory region of that type.

- **applyTransformer** We want to be able to apply transformations on an intrinsics definition and all of its arguments, so we need the method applyTransformer to take care of that. It will take any LMS expression x and will apply the transformer f on it. The result will be the transformed expression.

To illustrate the way we facilitate this class, we can investigate the implementation of an intrinsics that performs a store:

```scala
def _mm_store_ps[C[], U:Integral](
  mem_addr: Exp[C[Float]],
  a: Exp[_m128],
  mem_addrOffset: Exp[U])
  (implicit cont: Container[C]): Exp[Unit] = {
    cont.write(mem_addr)(MM_STORE_PS(mem_addr, a, mem_addrOffset))
}
```

In this example the parameter mem_addr corresponds to a pointer of floats. The offset is introduced by the last parameter that has the name of the pointer, followed by the suffix Offset. We use this offset strategy for all intrinsics that contain pointers. This is the major difference between the interface of the LMS intrinsics and the ones in the Intel Intrinsics Guide. The parameter a in this example is of type _m128 and refers to the packed values that will be stored in mem_addr. The container that determines the pointer implementation is passed as an implicit parameter. As we discussed in the previous section, stores should perform writes. Thus, the actual implementation of the write is delegated to this container. This is useful in cases where
an LMS write is insufficient for tracking all the dependencies. For example, when a pointer is cast to another pointer, the programmer should explicitly track the writes to both objects.

We should also illustrate how we perform mirroring of intrinsics that contain pointers, since the transformation process might require some knowledge of the extra dependencies of a pointer. The store introduced in the previous example should be mirrored in the following manner:

```scala
override def mirror[A:Typ](e: Def[A], f: Transformer)(implicit pos: SourceContext): Exp[A] = (e match {
  case iDef@MM_STORE_PS (mem_addr, a, mem_addrOffset) =>
    _mm_store_ps(
      iDef.cont.applyTransformer(mem_addr, f),
      iDef.cont.applyTransformer(a, f),
      iDef.cont.applyTransformer(mem_addrOffset, f))
...
})
```

As it is evident from the snippet above, we mirror the definition by calling the corresponding function and delegating the application of the transformer to the container that will specify the important operations regarding pointer manipulation.

### 4.7 Maven artefact

The intrinsics that we generated are hosted on the Maven central repository [5]. This is the dependency that needs to be added in order to use them:

```xml
<dependency>
  <groupId>ch.ethz.acl</groupId>
  <artifactId>lms-intrinsics_2.11</artifactId>
  <version>0.0.2-SNAPSHOT</version>
</dependency>
```

The project is still work in progress as the time of the writing, so it is published as a snapshot. Analogously, the following line would add the dependency to an SBT project:

```scala
dependencies += "ch.ethz.acl" %% "lms-intrinsics" % "0.0.2-SNAPSHOT"
```
The purpose of this chapter is to illustrate sample use cases of the intrinsics that we generated. Due to the fact that they are very suitable for problems that contain dense numeric computations, we decided that a subset of the original BLAS library [2] would be a fitting choice to achieve that. Our goal is to build multiple layers of abstraction and demonstrate that the intrinsics can be used for easy generation of code for different architectures and types by simply varying a set of parameters. We neither aim to implement the full BLAS specification, nor to offer a solution that has a higher performance than the original library. The following sections introduce a set of traits and methods that illustrate our strategy to build abstractions using the intrinsics.

5.1 Pointwise operations

Given that SIMD instructions are very suitable for parallelizing arithmetic operations, we are going to implement an interface for performing pointwise operations on two vectors of numbers. The operations that we support are addition, subtraction, multiplication and division. Moreover, we take multiple data types instruction set extensions into consideration.

We observed that all functions that perform vectorized arithmetic operations on two vectors have similar structure. They have a loop that is responsible for the vectorized computation and a scalar loop that finishes all elements that were not taken into account due to the divisibility of the register size. Thus, we implemented a base trait `PointwiseOperation` that contains the following method:
5. BLAS-like interface

```scala
def pointwiseOperation(
  step: Int,
  len: Exp[Int],
  sBody: Exp[Int] => Exp[Unit],
  vBody: Exp[Int] => Exp[Unit]): Rep[Unit] = {
  val vLen = infix_&(len, Const(-step))
  forloop(0, vLen, fresh[Int], Const(step), vBody)
  forloop(vLen, len, fresh[Int], 1, sBody)
}
```

The method has the following parameters:

- **step** This parameter determines the step size of the vectorized loop. This usually depends on how many numbers can be fit in a register simultaneously, depending on the underlying architecture. For example, an AVX register is 256 bits long, thus it can fit 8 floats that are 32 bits long.

- **len** This parameter corresponds to the length of the vectors the method operates on.

- **sBody** This is a function that takes a staged integer as input and returns a staged `Unit` result. It specifies the execution of the body of the scalar loop. The input of the function will be the bound variable of the loop.

- **vBody** Analogously to `sBody`, we need a parameter that will specify the computation in the vectorized loop. This is the purpose of the parameter `vBody`. It has the same type as `sBody`.

As shown in the code snippet above, the body of the method consists of only three instructions. First, we compute a length `vLen` that corresponds to the length `len`, but ignoring the last elements that would not fill a whole register. This is the length that is divisible by the step size `step`. Then, we introduce a helper LMS definition `forloop` that constitutes a more flexible implementation of the default for-loop of LMS. It takes five staged arguments. Setting the first one to 0 means that we want to start our loop from 0. With the second argument we determine the upper bound of the loop. In our case that means that we will iterate through all elements up to the element with index `vLen - 1`. The argument `fresh[Int]` means that we are going to use a fresh integer symbol as an iteration variable. With the fourth argument we state that we want a for-loop with step size depending on the parameter `step`. Finally, we insert the body `vBody` of the vectorized loop. After the vectorized loop follows a scalar one that finishes all the elements with indices from `vLen` to `len - 1`. We use a fresh iteration variable. The step size is 1 and we use the parameter `sBody` that specifies the body of the loop.

Now that we have a base method, we can use it to provide another layer of abstraction on top of it. One useful observation that we made was that
for given type and instruction set extension, the loads and stores will be the
same. Thus, only the actual arithmetic operation will vary. For example,
pointwise operations for AVX and type double rely on the following helper
method:

```scala
def doubleOpAvx[C[_], T:Numeric](
  v1: Rep[C[T]],
  v2: Rep[C[T]],
  dest: Rep[C[T]],
  len: Rep[Int],
  sBody: Exp[Int] => Exp[Unit],
  op: (Exp[__m256d], Exp[__m256d]) => Exp[__m256d],
  step: Int)(implicit cont: Container[C]) = {
  pointwiseOperation(step, len, sBody, (i: Exp[Int]) => {
    val r1 = _mm256_loadu_pd(v1, i)
    val r2 = _mm256_loadu_pd(v2, i)
    val res = op(r1, r2)
    _mm256_storeu_pd(dest, res, i)
  })
}
```

As it is evident from this code fragment, The signature now accepts two
staged pointers v1 and v2 of the generic higher-kinded type C[T], a destina-
tion pointer dest where to store the result, the length len, the body of the
scalar loop sBody, a parameter op that represents a function that will take
two arguments of type Exp[__m256d] and will return a result of the same
type, and a step size step.

We can now build yet another layer of abstraction on top of that that will
map a combination of an instruction set extension and a type with a specific
operation. For example, to connect the method that we previously defined
with the addition operation, we can use the trait VaddAVX:

```scala
trait VaddAVX extends PointwiseAVX with VaddSSE2 {

  def vaddAVX[C[_], T:Numeric](
    v1: Rep[C[T]],
    v2: Rep[C[T]],
    dest: Rep[C[T]],
    len: Rep[Int])
  (implicit m: Typ[T], cont: Container[C]): Rep[Unit] = {
    val sBody = (i: Exp[Int]) => {
      val r1 = cont.apply(v1, i)
      val r2 = cont.apply(v2, i)
      val res = r1 + r2
      cont.update(dest, i, res)
    }
  }
}
```
5. BLAS-like interface

```scala
m match {
  case _ if m <:< ManifestTyp(scala.Predef.manifest[Double]) =>
    doubleOpAvx(v1, v2, dest, len, sBody, _mm256_add_pd, 4)
  case _ if m <:< ManifestTyp(scala.Predef.manifest[Float]) =>
    floatOpAvx(v1, v2, dest, len, sBody, _mm256_add_ps, 8)
  case _ => vaddSSE2(v1, v2, dest, len)
}
```

The method `vaddAVX` has a way more generic signature: it accepts the representation of two numeric pointers `v1` and `v2`, a destination pointer `dest` for the result and the length of the vectors `len`. It then specifies a body for the scalar addition `sBody` that will be reused by all AVX implementations regardless of the type. Then the result is determined by the implicit parameter `m` which carries runtime information of the type `T`. The method `floatOpAvx` is analogous to the method `doubleOpAvx` that we introduced earlier, but takes float operations for AVX into account. The trait `VaddSSE2` has identical structure to `VaddAVX`, but will take into consideration all additions for SSE2. If there is no version that matches the type `m`, the computation in `vaddAVX` will be delegated to `vaddSSE2` that is implemented in `VaddSSE2`. In case the traits for all ISAs fail to provide a vectorized implementation, a default scalar version will be called.

Methods such as `vaddAVX` are useful, since they are responsible for optimal vectorized addition for all types, assuming the architecture is AVX. We define identical methods for subtraction, multiplication and division and then implement a base method that will choose the right implementation based on the instruction set extension, which is a variable. That way we are able to generate 54 combinations of operations, types and instruction set extensions.

5.2 Dot product

In the previous section we illustrated how we used abstraction to implement multiple versions of pointwise operations on two vectors. In this section we will focus on a single operation, which in this particular case is the dot product of two vectors, and will show how we can generate code for multiple data types and instruction set extensions.

Similarly to the case of pointwise operations, we observed that all versions of vectorized computations of dot product follow a similar structure. This structure is illustrated in our base dot product method `dotb`:
The method has three type parameters. The higher-kinded type \( C \) refers to our actual pointer implementation similar to our previous examples. The type \( T \) is again the type of the pointer that we represent. It should be a numeric type and runtime information for it should be present. The new type \( I \) that we introduced would be matched to the actual intrinsic type that we will use for the corresponding implementation of the dot product.

Now that we have abstracted the types that we will be working with, let us look into the actual implementation. First, we initialize the variable \( ac \) with a function set that is given as a parameter. Similarly to the pointwise example in the previous section, we compute a length \( vLen \) that will consider a range of elements, the computation of which will be vectorized. Afterwards we have a for-loop that is responsible for the vectorizable portion of the computation. It is interesting to note that in this example we use the functions load, store, mul and add that are given as a parameter and will determine how we should perform vectorized loads, stores, multiplications and additions respectively. After the variable \( ac \) is filled with partial dot products, we use the function hadd to compute a horizontal addition of those products and we store the result in the variable \( sum \). In the last part of the method we consider the remaining elements that were not taken into account due to
the divisibility of the length and add their dot product to \(\text{sum}\).

Now that we have this base method, we just need to specify how we perform the vectorized operations for the given vectors. For example, in order to provide a full implementation of a dot product for AVX and vectors of type double, we would have the following method call:

```python
dotb[C, Float, __m256](v1, v2, s1, s2, len,
_mm256_set1_ps(0.0F),
_mm256_loadu_ps,
_mm256_mul_ps,
_mm256_add_ps,
hadd256_ps, 8)
```

As shown above, we only need to specify which intrinsics to use for the vectorized computation, as well as the types. The arguments \(v1\) and \(v2\) correspond to the two vectors, \(s1\) and \(s2\) are their starting points and \(len\) is their length. The function \(\text{hadd256}\) is an implementation we provide for computing horizontal addition of a \(__m256\) vector. Using this abstraction we produced 6 vectorized versions for dot product and one default scalar one. We accomplished that by using a method that delegates the computation based on the instruction set architecture and the type of the vectors. It has the following signature:

```python
def dot[C[_], T:Numeric:Typ](
v1: Exp[C[T]],
v2: Exp[C[T]],
s1: Exp[Int],
s2: Exp[Int],
len: Exp[Int])
(implicit cont: Container[C]): Exp[T]
```

Similarly to many of the previous examples, the method has a type parameter \(C\) that determines the underlying pointer implementation, as well as a numeric type \(T\) that is the actual type of the pointer. The method has two parameters for the input vectors \(v1\) and \(v2\), their respective starting points \(s1\) and \(s2\) and their length \(len\).

## 5.3 Matrix-vector product

Now that we have a base method for computing the dot product of two vectors, we should build another layer of abstraction on top of it. One very representative example of a function from BLAS 2 is the matrix-vector product. What matrix-vector product in essence does is computing multiple dot products. Thus, we reuse the method \(\text{dot}\) that we introduced in the previous section. The resulting method for computing the matrix-vector product has the following implementation:
5.4. Matrix-matrix product

In this section we will use an identical approach for implementing a staged method for computing matrix-matrix product. This method has the following form:

```scala
def mmm[C[_], T:Numeric:Typ](
  mat1: Exp[C[T]],
  mat2: Exp[C[T]],
  res: Exp[C[T]],
  m: Exp[Int],
  k: Exp[Int],
  n: Exp[Int])
(implicit cont: Container[C]): Exp[Unit] = {
  val mat2T = transpose(mat2, k, n)
  forloop(0, m, fresh[Int], 1, (i: Exp[Int]) => {
    forloop(0, n, fresh[Int], 1, (j: Exp[Int]) => {
      cont.update(res, i * n + j, dot(mat1, mat2T, i * k, j * k, k))
    }
  })
}
```

As it is evident from the snippet above, the method is very concise and simple. It consists of a single loop that computes the dot product of each row of mat1 and the vector vec and stores the result in the corresponding entry of res. The method dot covers 7 implementations in total depending on the instruction set extension and the type, so the method matrixVectorProduct inherits this property as well. Thus, with only couple of lines we can generate multiple versions of matrix-vector product, tailored to our specific preference on ISA and type.
Given that matrix-vector product is a special case of matrix-matrix product, it is obvious that the method is similar to matrixVectorProduct introduced in the previous section. The difference of the signature is that it accepts two matrices mat1 and mat2 of sizes $m \times k$ and $k \times n$ respectively and the result is a matrix res of dimensions $m \times n$.

What the traditional matrix-matrix product algorithm does is computing all dot products of the rows of the first matrix and the columns of the second. Thus, we use a helper method transpose to get a transposed version mat2T of mat2. Once we have this transposed matrix, computing the desired dot products is straightforward. As a result, we again have 7 implementations of vectorized matrix-matrix product depending on the instruction set extension and the type of the entries.
In the previous sections we introduced the intrinsics to LMS and illustrated how we can build abstractions on top them. In this section we are going to investigate whether the generated code that relies on intrinsics provides better performance than an equivalent non-vectorized version. We take a look at one example with low and one with high operational intensity [43].

6.1 Benchmarking configuration

We ran our experiments under the following configuration:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache size</td>
<td>32 KB</td>
</tr>
<tr>
<td>L2 cache size</td>
<td>256 KB</td>
</tr>
<tr>
<td>L3 cache size</td>
<td>3 MB</td>
</tr>
<tr>
<td>Compiler</td>
<td>Apple LLVM version 7.0.2 (clang-700.1.81)</td>
</tr>
<tr>
<td>Java version</td>
<td>1.8.0_101-b13, JVM: 25.101-b13</td>
</tr>
<tr>
<td>Operating system</td>
<td>Mac OS X Yosemite 10.10.5</td>
</tr>
<tr>
<td>Compiler flags</td>
<td>-std=c99 -O3 -march=haswell</td>
</tr>
<tr>
<td>Intel Hyper-Threading</td>
<td>Off</td>
</tr>
<tr>
<td>Intel Turbo Boost</td>
<td>Off</td>
</tr>
</tbody>
</table>

Writing proper benchmarks on the Java Virtual Machine (JVM) is quite challenging. The JVM needs to perform many tasks such as class loading and garbage collection that might interfere with the measurements. Moreover, there are many optimizations that the JVM may apply to a component when it is executed in isolation. Thus, it is sensible to resort to a benchmarking framework for our evaluation that will take these issues into consideration and will isolate our experiments properly.
In order to execute benchmarks correctly, we used ScalaMeter [9], since it is a library that is specifically designed for experiments written in Scala. It spawns a new JVM instance for all our tests and ignores all samples that were intervened by Garbage Collection. Before all our benchmarks we perform 5000 warm up runs, then 1000 runs per size and for each size we take the median. The tests were run with a warm cache. We linked the C generated code that uses our intrinsics to the JVM using JNI.

6.2 Daxpy

The first example that we consider for our evaluation is the daxpy procedure that is a part of the original BLAS interface. The reason why we are interested in this algorithm is because it has low operational intensity, so we wanted to see how it performs when the runtime is heavily dominated by cache misses and data movement. Figure 6.1 illustrates the results:

The x-axis corresponds to the size of the vectors and the y-axis to the runtime of the implementations in milliseconds. Both axes are logarithmic. The type of the entries is float. We consider 3 versions in total. The lines “Daxpy Single Run” and “Daxpy JITed” both correspond to an implementation in Scala that relies on a single while-loop. The difference between the two is that “Daxpy JITed” was run multiple times according to the specification from the previous section, whereas “Daxpy Single Run” was run only once, thus it is an exception to the rule and is not properly optimized. The bold red line labeled “Daxpy LMS Intrinsics” corresponds to the version that
we generated using LMS and the intrinsics that we introduced. It has the very same structure as the Scala implementations, but the code was also vectorized using AVX intrinsics.

As shown in Figure 6.1, for sufficiently large sizes the JIT-ed Daxpy version is as fast as our implementation. For smaller sizes there is a minor difference which is due to JNI overhead. We investigated the assembly generated by the JIT compiler and observed that the code does not get vectorized. However, our version is not faster, since the computation is dominated by data movement.

The Daxpy implementation that was run once is orders of magnitudes slower than the other two versions.

### 6.3 Matrix-matrix multiplication

Given that vectorization is useful for problems that are dominated by arithmetic computations, we chose matrix-matrix multiplication for our evaluation, since it has considerably higher operational intensity than daxpy. The results are depicted in Figure 6.2:

![Figure 6.2: Runtime comparison of different implementations of matrix-matrix multiplication](image)

The x-axis corresponds to the size of one side of each matrix. We used square matrices, thus all sides are equal. The y-axis is logarithmic and denotes the runtime of the implementations in milliseconds. The type of the entries is `float`. Similar to the previous section, “MMM Single Run” and “MMM JITed” both correspond to a Scala implementation that relies on a triple
nested while-loop, where “MMM JITed” was run multiple times according to the specification from the previous section, while “MMM Single Run” was run only once, thus it is an exception to the rule and is not properly optimized. The bold red line labeled “MMM LMS Intrinsics” corresponds to the version that we generated using LMS and the intrinsics that we introduced. It has the very same structure as the Scala implementations, but in addition we also transpose the second matrix and vectorize the innermost loop.

The pattern for all sizes is consistent. The version that is not JIT-ed is around 10 times slower than its JIT-ed equivalent, whereas the JIT-ed Scala version is approximately 8 times slower than our implementation. This is expected, because in the case of matrix-matrix multiplication the computation is dominated by arithmetic operations. Due to the fact that AVX registers pack 8 floats simultaneously, the speedup with factor of 8 is not surprising. By investigating the assembly of the JIT-ed MMM implementation, we observed that the code does not get vectorized.
Now that we have introduced intrinsics for vectorization in LMS, there are multiple possibilities to facilitate them. All computations where data level parallelism might be exploited constitute a good domain that might make use of SIMD. Thus, as a future project one might find pertinent use cases where intrinsics will enhance the performance. In Chapter 5 we introduced a subset of BLAS procedures that we vectorized. One possible extension would be to provide a complete staged implementation of BLAS in LMS and introduce multiple domain-specific optimizations, so that the performance is on par with the original library.

Furthermore, in this work we address intrinsics that are supported by different Intel microarchitectures. As a future work it would be helpful to address other families of architectures, since some of them are ubiquitous in certain domains. For example, Acorn RISC Machine (ARM) architectures are the preferred choice for many mobile devices, thus providing facilities for generating vectorized code for them would be extremely beneficial, given how omnipresent mobile devices are. Moreover, the ARM Information Center website [1] provides a summary for all NEON intrinsics. Since we have introduced a methodology for integrating intrinsics within a compiler framework based on a specification, similar approach can be adopted for NEON intrinsics. This applies to the AMD’s 3DNow! extension as well [6].

Given that LMS is an embedded compiler framework, another viable direction that might be explored is automatic vectorization. There is a huge amount of literature on the topic and such feature is missing in LMS. The most common approach that most modern compilers adopt is SLP [27]. Its work mechanism is well known, so it can be ported into LMS with some effort. Although it traditionally works on successive instructions, it can be adopted for loop vectorization by relying on an algorithm for loop-unrolling. Moreover, various extensions like padding for matching non-isomorphic sequences [37] and addressing interleaved data [34] can be introduced.
Finally, since LMS is a framework designed for the implementation of domain-specific languages, a viable idea for future work would be to facilitate domain-specific optimizations that are based on the intrinsics. For example, certain intrinsics may be regrouped in order to reach best possible performance. Another more intricate option would be to match a group of intrinsics that has a specific structure and replace it with another group that has the same functionality but better performance.
Chapter 8

Conclusion

In this thesis we proposed a systematic methodology for generating intrinsics within a compiler framework based on a given specification. We illustrated our approach by introducing all Intel Intrinsics instructions to LMS. As part of the generation process we had to address various challenges. First, we had to come up with a strategy to remap all relevant C types to Scala types and ensure that our library is type-safe and that the correctness of the intrinsics is guaranteed by construction. Furthermore, given that the total number of intrinsics exceeds 5700, we had to address the limitations of the JVM regarding allowed bytecode size of the generated classes. Another issue was that there are some intrinsics that have side effects and others that perform reads and writes, so we had to make sure that those were captured correctly within our library. Finally, there is no equivalent of pure pointers in Scala, so we had to make sure that our implementation is not restricted to a particular pointer class. The resulting artefact was released on the Maven central repository.

Moreover, in order to demonstrate the usage and helpfulness of the intrinsics, we provided implementations for various procedures from the BLAS interface. We introduced methods for generating code for vectorized pointwise operations between two vectors that support multiple types, operations and instruction set extensions, resulting in 54 different versions. We adopted similar approach for developing an implementation for computing the dot product of two vectors. We reused this implementation to provide functionality for vectorized computation of matrix-vector product and matrix-matrix multiplication.

The benchmarks that we ran proved that our library can provide equal or better performance than the JVM. For problems with low operational intensity our library is as fast as a standard implementation written in Scala. However, for computations with high operational intensity our approach outperforms the JVM implementation by a factor that is determined by the
8. Conclusion

register size of the microarchitecture.
Bibliography


Decloration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

EXPLICIT SIMD INSTRUCTIONS INTO JVM USING LMS

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):
TOSKOV

First name(s):
IVAYLO

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Zurich, 12.06.2017

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.