SIMD Intrinsics on Managed Language Runtimes

Alen Stojanov
Department of Computer Science
ETH Zurich
Switzerland
astojanov@inf.ethz.ch

Ivaylo Toskov
Department of Computer Science
ETH Zurich
Switzerland
itoskov@student.ethz.ch

Tiark Rompf
Department of Computer Science
Purdue University
USA
tiark@purdue.edu

Markus Püschel
Department of Computer Science
ETH Zurich
Switzerland
pueschel@inf.ethz.ch

Abstract
Managed language runtimes such as the Java Virtual Machine (JVM) provide adequate performance for a wide range of applications, but at the same time, they lack much of the low-level control that performance-minded programmers appreciate in languages like C/C++. One important example is the intrinsics interface that exposes instructions of SIMD (Single Instruction Multiple Data) vector ISAs (Instruction Set Architectures). In this paper we present an automatic approach for including native intrinsics in the runtime of a managed language. Our implementation consists of two parts. First, for each vector ISA, we automatically generate the intrinsics API from the vendor-provided XML specification. Second, we employ a metaprogramming approach that enables programmers to generate and load native code at runtime. In this setting, programmers can use the entire high-level language as a kind of macro system to define new high-level vector APIs with zero overhead. As an example use case we show a variable precision API. We provide an end-to-end implementation of our approach in the HotSpot VM that supports all 5912 Intel SIMD intrinsics from MMX to AVX-512. Our benchmarks demonstrate that this combination of SIMD and metaprogramming enables developers to write high-performance, vectorized code on an unmodified JVM that outperforms the auto-vectorizing HotSpot just-in-time (JIT) compiler and provides tight integration between vectorized native code and the managed JVM ecosystem.

CCS Concepts · Computer systems organization → Single instruction, multiple data; · Software and its engineering → Virtual machines; Translator writing systems and compiler generators; Source code generation; Runtime environments;

Keywords SIMD instruction set, Managed Languages, JVM, Scala, Staging, Metaprogramming

ACM Reference Format:

1 Introduction
Managed high-level languages are designed to be portable and to support a broad range of applications. For the programmer, the price is reduced access to detailed and low-level performance optimizations. In particular, SIMD vector instructions on modern architectures offer significant parallelism, and thus potential speedup, but neither languages like Java, JavaScript, Python, or Ruby, nor their managed runtimes provide direct access to SIMD facilities. This means that SIMD optimizations, if available at all, are left to the virtual machine (VM) and the built-in just-in-time (JIT) compiler to carry out automatically, which often leads to suboptimal code. As a result, developers may be pushed to use low-level languages such as C/C++ to gain access to the intrinsics API. But leaving the high-level ecosystem of Java or other languages also means to abandon many high-level abstractions that are key for the productive and efficient development of large-scale applications, including access to a large set of libraries.

To reap the benefits of both high-level and low-level languages, developers using managed languages may write low-level native C/C++ functions, that are invoked by the managed runtime. In the case of Java, developers could use the Java Native Interface (JNI) to invoke C functions with specific naming conventions. However, this process of dividing
the application logic between two languages creates a significant gap in the abstractions of the program, limits code reuse, and impedes clear separation of concerns. Further, the native code must be cross-compiled ahead of time, which is an error-prone process that requires complicated pipelines to support different operating systems and architectures, and thus directly affects code maintenance and refactoring.

To address these problems, we propose a systematic and automated approach that gives developers access to SIMD instructions in the managed runtime, eliminating the need to write low-level C/C++ code. Our methodology supports the entire set of SIMD instructions in the form of embedded domain-specific languages (eDSLs) and consists of two parts. First, for each architecture, we automatically generate ISA-specific eDSLs from the vendor’s XML specification of the SIMD intrinsics. Second, we provide the developer with the means to use the SIMD eDSL to develop application logic, which automatically generates native code inside the runtime. Instead of executing each SIMD intrinsic immediately when invoked by the program, the eDSLs provide a staged or deferred API, which accumulates intrinsic invocations along with auxiliary scalar operations and control flow, batches them together in a computation graph, and generates a native kernel that executes them all at once, when requested by the program. This makes it possible to interleave SIMD intrinsics with the generic language constructs of the host language without switching back and forth between native and managed execution, enabling programmers to build both high-level and low-level abstractions, while running SIMD kernels at full speed.

This paper makes the following contributions:

1. We present the first systematic and automated approach that supports the entire set of SIMD instructions, automatically generated from the vendor specification, in a managed high-level language. The approach is applicable to other low-level languages, provided support for native code binding in the managed high-level language.
2. In doing so, we show how to use metaprogramming techniques and runtime code generation to give back low-level control to developers in an environment that typically hides architecture-specific details.
3. We provide an end-to-end implementation of our approach within the HotSpot JVM, which provides access to all Intel SIMD intrinsics from MMX to AVX-512.
4. We show how to use the SIMD eDSLs to build new abstractions using host language constructs. Programmers can use the entire managed language as a form of macro system to define new vectorized APIs with zero overhead. As an example, we present a “virtual ISA” for variable precision arithmetic.
5. We provide benchmarks that demonstrate significant performance gains of explicit SIMD code versus code auto-vectorized by the HotSpot JIT compiler.

Our work focuses on the JVM and Intel SIMD intrinsics functions, but would equally apply to other platforms. For the implementation of computation graphs and runtime code generation, we use the LMS (Lightweight Modular Staging) compiler framework [20].

2 Background

We provide background on intrinsics functions, JVMs, and the LMS metaprogramming framework that we use.

2.1 Intrinsics

Intrinsics are compiler-built-in functions that usually map into a single or a small number of assembly instructions. During compilation, they are inlined to remove calling overhead. This way they provide the programmer with assembly-like functionality, without having to worry about register allocation and instruction scheduling. SIMD intrinsics give access to data parallel instructions in vector ISAs, such as NEON on ARM processors, or the SSE and AVX families on Intel.

We focus on the x86 architecture and the associated SIMD intrinsics that are available in modern C/C++ compilers, such as GCC, Clang/LLVM, and Intel ICC. Specifically, these include the following ISAs:

- MMX - operating on 64-bit wide registers; provides integer operations only.
- SSE / SSE2 / SSE3 / SSSE3 / SSE4.1 / SSE4.2 - operating on 128-bit wide registers; provides integer, 32-bit, and 64-bit floating point operations, string operations and cache and memory management operations.
- AVX / AVX2 - ISAs that expand the SSE operations to 256-bit wide registers and provide extra operations for manipulating non-contiguous memory locations.
- FMA - an extension to SSE and AVX ISAs to provide fused multiply add operations.
- AVX-512 - extends AVX to operate on 512-bit registers and consists of multiple parts called F / BW / CD / DQ / ER / IFMA52 / PF / VBMI / VL.
- KNC - the first production version of Intel’s Many Integrated Core (MIC) architecture that provides operations on 512-bit registers.

Additionally, we also include:

- SVML - an intrinsics short vector math library, built on top of the ISAs mentioned above.
shows the number of VM, as it is tuned to maximize peak operating speed. The
provides JIT compilation of Java bytecode as a black box. The
developer has no control over, nor receives any feedback on the
compilation phases except through coarse-grained command-line and debug options [16]. There are two flavors of
the VM: a client mode focused on latency, and a server
mode tuned for throughput. We only focus on the Server
VM, as it is tuned to maximize peak operating speed. The Server VM offers a tiered compilation of bytecode using the
C1 and C2 compilers. C1 is a fast, lightly optimizing bytecode
compiler, while C2 performs more aggressive optimizations. When JVM applications are started, the HotSpot VM starts
interpreting bytecode. It detects computation-intensive hot
spots in the code via profiling, and proceeds to compile the
bytecode of frequently used functions with C1. Once further
thresholds are reached, functions may be compiled using
C2. C2 supports autovektorization, using Superword Level
Parallelism (SLP) [11]. SLP detects groups of isomorphic
instructions and replaces them with SIMD instructions, which
results in a lightweight vectorization. The SLP approach is
limited and cannot optimize across loop iterations, nor can
it detect idioms such as reductions.

<table>
<thead>
<tr>
<th>ISA</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMX</td>
<td>124</td>
</tr>
<tr>
<td>SSE</td>
<td>154</td>
</tr>
<tr>
<td>SSE2</td>
<td>236</td>
</tr>
<tr>
<td>SSE3</td>
<td>32</td>
</tr>
<tr>
<td>SSE41</td>
<td>61</td>
</tr>
<tr>
<td>SSE42</td>
<td>19</td>
</tr>
<tr>
<td>AVX</td>
<td>188</td>
</tr>
<tr>
<td>AVX2</td>
<td>191</td>
</tr>
<tr>
<td>AVX-512</td>
<td>3857</td>
</tr>
<tr>
<td>FMA</td>
<td>32</td>
</tr>
<tr>
<td>KNC</td>
<td>601</td>
</tr>
<tr>
<td>SVML</td>
<td>406</td>
</tr>
</tbody>
</table>

2.2 Java Virtual Machines

There are many active implementations of the JVM including
the open source IBM V9 [8], Jikes RVM [4], Maxine [29],
JRockit [15], and the proprietary SAP JVM [24], CEE-J [26],
JamaicaVM [25]. HotSpot remains the primary reference
JVM implementation that is used by both Oracle Java and
OpenJDK. Each of the JVM implementations provides support
for Java Standard Edition or Micro Edition, tailored for a
particular need: either a particular target machine or microar-
chitecture, embedded systems, operating system, provides
additional garbage collector, resource control or parallelism
model. However, none of the active JVM implementation
provides any support for explicit vectorization nor intrinsics,
nor permits inlining of assembly code directly in the Java
source due to portability issues.

The HotSpot JVM, which is the focus of this study, pro-
vides JIT compilation of Java bytecode as a black box. The
developer has no control over, nor receives any feedback on the
compilation phases except through coarse-grained command-line and debug options [16]. There are two flavors of
the VM: a client mode focused on latency, and a server
mode tuned for throughput. We only focus on the Server
VM, as it is tuned to maximize peak operating speed. The Server VM offers a tiered compilation of bytecode using the

<table>
<thead>
<tr>
<th>ISA</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMX</td>
<td>124</td>
</tr>
<tr>
<td>SSE</td>
<td>154</td>
</tr>
<tr>
<td>SSE2</td>
<td>236</td>
</tr>
<tr>
<td>SSE3</td>
<td>32</td>
</tr>
<tr>
<td>SSE41</td>
<td>61</td>
</tr>
<tr>
<td>SSE42</td>
<td>19</td>
</tr>
<tr>
<td>AVX</td>
<td>188</td>
</tr>
<tr>
<td>AVX2</td>
<td>191</td>
</tr>
<tr>
<td>AVX-512</td>
<td>3857</td>
</tr>
<tr>
<td>FMA</td>
<td>32</td>
</tr>
<tr>
<td>KNC</td>
<td>601</td>
</tr>
<tr>
<td>SVML</td>
<td>406</td>
</tr>
</tbody>
</table>

2.3 Lightweight Modular Staging

Lightweight Modular Staging (LMS) [20, 21] is a framework
for runtime code generation and for building compilers for
embedded DSLs in Scala. LMS makes pervasive use of op-
erator overloading to make code generation blend in with
normal programming. The core abstraction is a type con-
structor Rep[T] that marks code expressions. For example,
excuting a + b where a and b are two Rep[Int] expressions
will create a program expression that represents the addi-
tion a’ + b’, where a’ and b’ are the program expres-
sions a and b evaluate to. This form of operator overloading is ex-
tended to if/else expressions and other built-in constructs
[2, 19]. The combined program expression can be unparsed
to source code, in this paper to C or LLVM code, compiled
dynamically, and loaded into the running JVM.

3 Intrinsics in the JVM

In this section, we present our two-tier approach for making
the Intel SIMD intrinsics available in the JVM. First we automatically generate SIMD eDSLs, each implemented as a
Scala class that corresponds to one of the 13 vector ISAs in
Figure 1b. Then, we show how to use these eDSLs to generate

<table>
<thead>
<tr>
<th>ISA</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMX</td>
<td>124</td>
</tr>
<tr>
<td>SSE</td>
<td>154</td>
</tr>
<tr>
<td>SSE2</td>
<td>236</td>
</tr>
<tr>
<td>SSE3</td>
<td>32</td>
</tr>
<tr>
<td>SSE41</td>
<td>61</td>
</tr>
<tr>
<td>SSE42</td>
<td>19</td>
</tr>
<tr>
<td>AVX</td>
<td>188</td>
</tr>
<tr>
<td>AVX2</td>
<td>191</td>
</tr>
<tr>
<td>AVX-512</td>
<td>3857</td>
</tr>
<tr>
<td>FMA</td>
<td>32</td>
</tr>
<tr>
<td>KNC</td>
<td>601</td>
</tr>
<tr>
<td>SVML</td>
<td>406</td>
</tr>
</tbody>
</table>
Table 2. Type mappings between JVM and C/C++ types.

<table>
<thead>
<tr>
<th>JVM Types</th>
<th>C/C++ Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float</td>
<td>float</td>
</tr>
<tr>
<td>Double</td>
<td>double</td>
</tr>
<tr>
<td>Byte</td>
<td>char</td>
</tr>
<tr>
<td>Short</td>
<td>int16_t</td>
</tr>
<tr>
<td>Int</td>
<td>int32_t</td>
</tr>
<tr>
<td>Long</td>
<td>int64_t</td>
</tr>
<tr>
<td>__m256d</td>
<td>128-bit float</td>
</tr>
<tr>
<td>__m512d</td>
<td>256-bit float</td>
</tr>
<tr>
<td>__m128d</td>
<td>128-bit integer</td>
</tr>
<tr>
<td>__m256i</td>
<td>128-bit integer</td>
</tr>
<tr>
<td>__m512i</td>
<td>256-bit integer</td>
</tr>
<tr>
<td>__m128i</td>
<td>128-bit integer</td>
</tr>
</tbody>
</table>

3.1 Type System for SIMD Intrinsics in the JVM

The JVM has no notion of SIMD vector types, thus we build abstract classes to mark the type of DSL expressions that represent SIMD intrinsics functions in LMS:

```scala
case class Def[T] // define function type
case class Exp[T] // define expression type
```

 SIMD intrinsics functions take primitive arguments that correspond to low-level C/C++ primitive types. The primitive types in the JVM exhibit a fixed width, and therefore a direct mapping can be established with C/C++ primitives. Some intrinsics however, require the use of unsigned types that are not supported natively in the JVM:

```scala
unsigned int __mm_crc32_u16 (unsigned int, unsigned short)
```

To mitigate this problem, we use the Scala Unsigned package, which implements unsigned types and operations on top of the signed types available in the JVM. Table 2 shows the type mapping between the 12 primitives, which in most cases is straightforward, except for JVM char that maps to int16_t to support UTF-8. Arrays of primitive types in the JVM and C/C++ code are isomorphic and both represent continuous memory space of a certain primitive type. Therefore Array[T] maps to a memory pointer T* in the low-level SIMD intrinsics.

3.2 Automatic Generation of ISA-specific eDSLs

LMS provides a relatively simple interface to define eDSLs, but adding more than 5000 functions by hand would be tedious and error prone. Our approach generates the LMS eDSLs automatically from the XML specification provided by the Intel Intrinsics Guide; these are then packed as a jar file that is later published in the Maven Central Repository [5] for deployment. At the time of writing, we use the latest version of the intrinsics specifications stored as data-3.3.16.xml file, and build the generator such that it anticipates future extensions in the specifications. Figure 1 shows a high-level overview of the generation process, which we explain step-by-step next.

**Figure 1.** Generating SIMD intrinsics eDSLs from vendor specification.

```xml
<intrinsic rettype='__m256d' name='_mm256_add_pd'>
  <type>Floating Point</type>
  <CPUID>AVX</CPUID>
  <category>Arithmetic</category>
  <parameter varname='a' type='__m256d'/>
  <parameter varname='b' type='__m256d'/>
  <description>
    Add packed double-precision (64-bit) floating-point elements in "a" and "b", and store the results in "dst".
  </description>
  <operation>
    FOR j := 0 to 3
      i := j*64
    ENDFOR
    dst[MAX:256] := 0
  </operation>
  <instruction form='ymm, ymm, ymm'/>
  <header>immintrin.h</header>
</intrinsic>
```

**Figure 2.** XML specification of the _mm256_add_pd intrinsic.
function, return type, ordered list of each argument of the function with the corresponding type, CPUID parameter, that correspond to the ISA set, and a category parameter.

**Generate ISA-specific DSL in LMS.** For each intrinsic function we implement four building blocks that define the dDSL. These are represented in terms of implementation classes provided by the LMS framework. The classes Def[T] and Exp[T] together define a computation graph. Subclasses of Def[T] implement graph nodes that represent individual computations, e.g., Plus(a, b). Here, a and b are values of type Exp[T]; either constants Const(.) or symbols Sym(id) that refer to other graph nodes through a numeric index id.

The four necessary building blocks are as follows:

1. Definition of the intrinsic function represented as a subclass of Def[T].
2. Implicit conversion from expression Exp[T] to definition Def[T], looking up a computation node given a symbolic reference.
3. Mirroring function that converts a Def[T] into expression Exp[T], potentially applying a transformation.
4. Unparsing routine that converts each Def[T] into C/C++ string.

To complete the first part, we define IntrinsicsDef[T], an abstract class that each intrinsics definition will inherit:

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

Then for each intrinsic function, we define a Scala case class that corresponds to the intrinsics function’s name, its input arguments and return type. Each case class contains the category, the type of the intrinsics and performance informations when available. Additionally, we also include the header where the C/C++ intrinsics is defined:

```scala
case class MM256_ADD_PD(
  a: Exp[_m256d], b: Exp[_m256d]) {
  val category = List(Arithmetic)
  val intrinsicType = List(FloatingPoint)
  val performance = Map.empty[MicroArchType, Performance]
  val header = "immintrin.h"
}
```

With the current definition we allow a particular intrinsics to pertain to several categories. The header information gives us the control to include the correct header when unparsing the code to C/C++ code. Performance information is included but is not used in the staging process.

Next we generate Scala code for the implicit conversion from intrinsics expressions to definitions. This routine is essential in LMS, as it provides automatic conversion of the staged code into static single assignment (SSA) form. In most cases it is sufficient to rely on the Scala compiler to automatically perform the implicit conversion:

```scala
def _mm256_add_pd(a: Exp[_m256d], b: Exp[_m256d]) =
  MM256_ADD_PD(a, b)
```

The LMS framework supports DSL transformations by substitution. Once a substitution is defined, LMS creates new definitions. However, when no substitution is available, a definition has to be converted to an expression through a routine of mirroring that converts a Def[T] back to Exp[T], potentially creating a new definitions for subexpressions as part of the transformation:

```scala
override def mirror[A: Type](e: Def[A], f: Transformer)(implicit pos: SourceContext): Exp[A] = (e match {
  case MM256_ADD_PD(a, b) =>
    _mm256_add_pd(f(a), f(b))
    // ... a lot more patterns to match
  case _ => super.mirror(e, f)
})
```

Once code is generated for all these routines and for each intrinsic, the final step is to generate code to perform the unparsing of the DSL into C code. The unparsing routine is done similarly to the mirroring routine, by pattern matching each DSL definition to produce the corresponding C expression:

```scala
override def emitNode(s: Sym[Any], r: Def[Any]) = r match {
  case iDef@MM256_ADD_PD(a, b) =>
    headers += iDef.header
    emitValDef(s, "_mm256_add_pd($a, $b)")
    // ... a lot more patterns to match
  case _ => super.emitNode(s, rhs)
}
```

**Infer intrinsic mutability** As mentioned before, when code is generated for the implicit conversion of an intrinsics expression to the intrinsics definition, we can rely on the Scala compiler to match the correct implicit method. This works correctly for immutable expressions, but not all intrinsics are immutable. For example, each intrinsics that loads and stores from/to memory creates effects that have to be handled by LMS. The semantic of these effects is essential in scheduling the DSL.

To resolve this problem, we use the category information of each intrinsics (see Figure 2), and implement a conservative heuristic to generate the effects:

- Each time an intrinsics is discovered with a load category, we generate a read effect on each argument that is a memory location.
- Each time an intrinsics is discovered with a store category, we generate a write effect on each argument that is a memory location.

For example, an AVX load of 4 doubles has the form:

```scala
def _mm256_load_pd[A, U: Integral](
  // ... a lot more patterns to match
  emitValDef(s, "_mm256_load_pd($a, $b)")
  //... a lot more patterns to match
}
```
Start developing SIMD code

1. Implement a native function placeholder
2. Mixin one or several ISA-specific eDSLs
3. Implement the SIMD staged function
4. Call compile to generate native code

Start the Java Virtual Machine (JVM)

Detect available C/C++ compilers
Inspect the system through CPUID
Infer available ISAs and compiler flags
LMS: remove abstraction & generate C code

Compile and link the code to the JVM
Use high performance SIMD code

The heuristics is invoked on each intrinsics that performs loads, stores, maskstores, maskloads, gather, scatters and other intrinsics that perform memory-related operations.

Split each ISA specific DSL into subclasses The JVM has a hard limit of 64KB on the size of each method, which is an obstacle in generating the unparsing and mirroring routines for large ISA, such as AVX-512 or KNC. To avoid this obstacle, and still keep the LMS design pattern, we decided to split the ISA specific DSLs into subclasses that inherit each other.

3.3 Developing Explicitly Vectorized Code in the JVM Using SIMD eDSLs

Figure 3 gives a high-level overview of how to use explicit vectorization in the JVM. The process consists of two parts: compile-time tasks, done by the high-performance code developer, and runtime tasks that are done automatically by LMS and our compiler pipeline. Specifically, the compile-time tasks of the developer comprise four steps:

1. Implement a native function placeholder that will represent the vectorized code.
2. Create a DSL instance by instantiating one or mixing several ISA-specific eDSLs.
3. Implement the SIMD logic as a staged function.
4. Call the provided compile routine to generate, compile and link the code in the JVM.

Figure 4. A complete implementation of the BLAS 1 routine SAXPY in the JVM using AVX and FMA SIMD intrinsics.

After the four steps are completed, and the JVM program is started, the compiler pipeline is invoked with the compile routine. This will perform system inspection, search for available compilers and opportunistically pick the optimal compiler available on the system. In particular, it will attempt to find icc, gcc or llvm/clang. After a compiler is found, the runtime will determine the target CPU, as well as the underlying micro-architecture to derive available ISAs. This allows us to have full control over the system, as well as to be able to pick the best mix of compiler flags for each compiler.

Once this process is completed, the user-defined staged function is executed, which assembles a computation graph of SIMD instructions. From this computation graph, LMS generates vectorized C code. This code is then automatically compiled as a dynamic library with the set of derived compiler flags, and linked back into the JVM. To link the native code into the JVM, JNI requires the C functions header to contain the Java_prefix, followed by package name, class
name and name of the native function. The compile routine automates this process using JVM reflection and some lightweight use of Scala macros. By this automation, we ensure the interoperability between the native function and the staged function, creating code robust to modifications and refactoring and eliminating the need for the developer to recompile the native code each time major code revision are performed on the low-level code or the class container.

Figure 4 illustrates a complete and self-contained implementation of a BLAS 1 routine called SAXPY [12], which computes \( y = y + ax \) for given vectors \( x \), \( y \) and scalar \( a \). The expression for \( \text{loop}(\ldots) \) creates a staged loop in the LMS computation graph.

3.4 Evaluation

To assess the viability of our approach we consider two ubiquitous kernel functions: the aforementioned SAXPY and matrix multiplication.

**Experimental setup.** We perform the tests on a Haswell enabled processor Intel Xeon CPU E3-1285L v3 3.10GHz with 32GB of RAM, running Debian GNU/Linux 8 (jessie), kernel 3.16.43-2+deb8u3. The available compilers are gcc 4.9.2-10 and Intel icc 17.0.0. The installed JVM is HotSpot 64-Bit Server 25.144-b01, Supporting Java 1.8. To avoid the effects of frequency scaling and resource sharing on the measurements, Turbo Boost and Hyper-Threading are disabled.

We use ScalaMeter [17] to perform the benchmarks. To obtain precise results, we select a pre-configured benchmark that forks a new JVM virtual machine and performs measurements inside the clean instance. The new instance has a compilation threshold of 100 (-XX:CompileThreshold=100) and we perform at least 100 warm-up runs on all test cases to trigger the JIT compiler. Each test case is performed on a warmed cache. Tests are repeated 30 times, and the median of the runtime is taken. We show the results as performance, measured in flops per cycle.

To inspect the JIT compilation of the HotSpot JVM, we use -XX:UnlockDiagnosticVMOptions that unlocks the diagnostic JVM options and -XX:CompileCommand=print to output the generated assembly. In all test cases we observe full-tiered compilation starting from the C1 compiler to the last phase of the C2 compiler. For a fair comparison between the JVM and our generated intrinsics code, we consider the C2 compiled version of the bytecode only, excluding the JIT warm-up time and the LMS generation overhead.

**SAXPY** We compare the generated SAXPY vector code, shown in Figure 4, against an equivalent Java implementation:

```java
public class JSaxpy {
    public void apply(float[] a, float[] b, float s, int n) {
        for (int i = 0; i < n; i++)
            a[i] = b[i] * s;
    }
}
```

Figure 6a shows the performance comparison. First we note the similarity in performance, which is not surprising since SAXPY has low operational intensity and the simplicity of the code enables efficient autovectorization. Indeed, the assembly diagnostics confirms this but reveals that the JVM only uses SSE whereas our staged version uses AVX and FMA, which explains the better performance for larger sizes.

For small sizes that are L1 cache resident the Java implementation does better. This is because JNI methods are not inlined and incur additional cost to be invoked.
**Matrix-matrix multiplication (MMM).** For the second benchmark we chose MMM, which has a high operational intensity and is known to benefit from various optimizations such as blocking and vectorization [32]. We consider three versions. The first is a standard Java implementation of a triple MMM loop. The two other versions are a blocked version of MMM, with block size of 8, the first implemented in Java, and the second implemented using AVX intrinsics in Scala. For simplicity, we assume that the matrix has size $n = 8k$, and provide the implementation in Figure 5.

Our Scala implementation uses the SIMD intrinsics with high level constructs of the Scala language, including pattern matching (lines 5, 10, 19, 20, etc), lambdas (lines 4, 10, 34), Scala collections (lines 4, 34, etc), closures (line 45) and others that are not available in low-level C code. Once LMS removes the abstraction overhead, the MMM function results in a high-performance implementation. The performance comparison in Figure 6b shows that the use of explicit vectorization through SIMD intrinsics can offer improvements up to 5x over the blocked Java implementation, and over 7.8x over the baseline triple loop implementation.

The assembly analysis shows that the C2 compiler will unroll the hot loops in both Java versions, but does generate SIMD instructions, which explains the low performance.

**Automatic SIMD eDSL generator.** The predecessor of the Intel Intrinsics Guide web application was a Java application sharing the same name. The older versions of both Java and the web application contained older version of the intrinsics specifications, e.g., without AVX-512. However, Intel does not offer these versions, and continuously updates the XML specifications, improving the description / performance of each intrinsic function.

Using tools such as the Wayback Machine, a digital archive that mirrors web-site states at a given date, we were able to salvage older, pre-captured iterations of the intrinsics specifications, shown in Table 3. Then we instructed our eDSL generator to re-generate each ISA-specific eDSL.

Our results show that our eDSL generator is robust towards minor changes on the XML specifications, being able to retrospectively generate eDSLs for recent years. We believe that if Intel uses the same XML schema for new releases, our generator should be robust to new ISA updates, as long as the new ISA has similar properties than its predecessor.

### 3.5 Limitations

Our approach provides low-level control for performance optimizations to the Java developer but comes at a price. We discuss a number of technical issues that would be good to resolve to further improve ease-of-use and maintainability.

Currently, there is no mechanism to ensure the isomorphism between the native function placeholder and the staged function. As a result, it is the responsibility of the developer to define this isomorphic relation upon compile time. The current use of Scala macros makes the code robust in terms of refactoring and modifications, which is quite convenient compared to manually maintaining isomorphism between native C/C++ and JVM code. A more diligent use of Scala macros could potentially resolve this problem and ensure complete isomorphic binding of JNI and staged functions.

LMS does not provide any mechanism to deal with exceptions such as segfaults from generated code. Therefore it is the responsibility of the developer to write valid SIMD code. LMS is also not optimized for fast code generation,

---

**Figure 6. Performance analysis: Java implementation vs LMS intrinsics generated code.**

**Table 3. Intel Intrinsics Guide XML specifications.**

<table>
<thead>
<tr>
<th>Specification</th>
<th>Date</th>
<th>Specification</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>data-3.3.11.xml</td>
<td>27.07.2015</td>
<td>data-3.4.xml</td>
<td>07.09.2017</td>
</tr>
</tbody>
</table>
which might result in an overhead surpassing the HotSpot interpretation speed when used to generate functions that are computationally light.

Another limitation is a consequence of the complex memory model in the HotSpot JVM. Once arrays are used in the native code, GetPrimitiveArrayCritical must be invoked to obtain the memory space of the array. Depending on the state of the garbage collector (GC), the array might end up on different segments on the heap, which could result in a copy once the native code tries to access the memory space, or the JVM could decide to temporary disable the GC. Although we did not experience an array copy in any test case performed, we believe that the use of LMS intrinsics is best suited for compute-bound problems, where the copy overhead can be leveraged by the fast runtime of SIMD instructions upon each JNI invocation. Some of the issues with JVM arrays can be avoided by using Java NIO buffers or off-heap memory allocated with the sun.misc.Unsafe package.

4 Build Your Own Virtual ISA

In the previous section, we demonstrated the use of SIMD intrinsics in developing high performance code, based on high-level constructs of the Scala language. However, with the use of metaprogramming and staging provided by LMS, we can also use the SIMD intrinsics to build new low-level abstractions and provide a functionality similar to the SVML short vector math library that is typically implemented by low-level C/C++ compilers. As an example, we build abstractions for low-precision arithmetic and, in particular, building blocks for the stochastic gradient descent (SGD) algorithm.

SGD is currently among the most popular algorithms in machine learning, used for training neural networks [1, 22]. It consists of two main building blocks: a dot-product operator and a scale-and-add operator. The use of low precision is an important optimization in SGD for deep learning as it reduces both computation time and data movement for increased performance and efficiency [30].

In this section we build a virtual variable-precision ISA that implements the dot product operator, operating on arrays of 32, 16, 8 and 4-bit precision. For 32 and 16-bit we use floating point, which is natively supported by the hardware; for the lower precision formats, we use quantized arrays [30]. Quantization is a lossy compression technique that maps continuous values to a finite set of fixed-bit-width numbers. For a given vector \( \mathbf{v} \) of size \( n \) and precision of \( b \) bits, we first derive a factor \( s_b \) that scales the vector elements \( v_i \) into the representable range:

\[
s_b = \frac{2^{b-1} - 1}{\max_{i \in [1, n]} |v_i|}
\]

The scaled \( v_i \) are then quantized stochastically:

\[
v_i \rightarrow [v_i \cdot s_b + \mu]
\]

where \( \mu \) is drawn uniformly from the interval \((0, 1)\). With this, a quantized array consists of one scaling factor and an array of quantized \( b \)-bit values.

4.1 Implementation

32-bit. For the 32-bit version, we use the built-in hardware support for floating point. In Java, we can only express scalar (non-vectorized) code to multiply / add values; using our LMS intrinsics we have access to AVX2 and FMA.

16-bit. For the 16-bit version we use a half-precision floating point format, available as a separate ISA extension called FP16C that provides instructions to convert a 32-bit float into 16-bit float, and vice-versa. We use these instructions to load and store the data in 16-bit format, and perform computations on the 32-bit format. In Java, there is no access to half-precision floating point; thus, instead, we quantize the values as shown before to type short.

8-bit. We base our 8-bit version on Buckwild! [23]. Both LMS intrinsics and the Java implementation operate on quantized values, using a scaling factor and an array of type byte to hold the 8-bit two’s complements values.

4-bit. We base this version on the analysis in the ZipML framework [33]. The values are not two’s complement, but sign-bit followed by the base in binary format, and stored as pairs inside the 8-bit values of a byte array.

Scala implementation. In Scala, we can abstract the precision as a number that reflects the bit length of the format, and provide two virtual intrinsics functions:

```scala
def dot_ps_step(b: Int): (_: Int) => Int = acc => mm256_setzero_ps();
val increment = dot_ps_step(8); forloop(0, len, fresh[Int], increment, i => {
    acc = mm256_add_ps(acc, mm256_setzero_ps());
    result = mm256_add_ps(acc, mm256_setzero_ps());
}
val result = mm256_add_ps(acc, mm256_setzero_ps());
```

```scala
def dot_ps_step(bits: Int): (_: Int, void*: x, void*: y) => Int = acc => mm256_setzero_ps();
dot_ps_step returns the number of elements processed by dot_ps for a given bit length. For example, in the case of 32, 16 and 8-bit versions, 32 elements are processed at a time and in the case of the 4-bit, 128 elements at a time.
```

Finally, the resulting dot product with variable precision is a sum reduction of 8 floats stored in the acc variable:

```scala
    var acc = mm256_setzero_ps();
    val increment = dot_ps_step(bits);
    forloop(0, len, fresh[Int], increment, i => {
        acc = mm256_add_ps(acc, mm256_setzero_ps());
}
    reduce_sum(acc)
}
```
shows the obtained results. Our 4-bit implementation outperforms HotSpot by a factor of up to 40x, the 8-bit up to 9x, the 16-bit up to 4.8x, and the 32-bit version up to 5.4x. There are several reasons for the speedups obtained with the use of SIMD intrinsics. In the 32-bit case, we see the limitation of SLP to detect and optimize reductions. In the 16-bit, there is no way to obtain access in Java to an ISA such as FP16C. And in the 8-bit and 4-bit case, Java is severely outperformed since it does type promotion when dealing with integers. However, the largest speedup of 40x in the 4-bit case is due to the domain knowledge used for implementing the dot product, that the HotSpot compiler cannot synthesize with a lightweight autovectorization such as SLP.

5 Related Work

We review different lines of related work.

Explicit vectorization in the JVM. The first approach to expose data parallelism was the implementation of the Java Vectorization Interface (JVI) as part of the Jitrino JIT compiler [13]. JVI is designed as an abstract vector interface that provides set of methods as vector operators. This methods are later compiled to different vector interface, such as SSE and AVX. The approach offers competitive results in some case, but is limited in the SIMD instructions it supports and subsequent iterations of post-AVX ISAs. Similarly to JVI, Oracle has ongoing research developing cross-platform APIs that can leverage SIMD instructions. Implemented as part of an experimental JVM called Panama [9], SIMD instructions are used in immutable vector types, parameterized by element type and size. Similarly to JVI, Panama also suffers from limited support of vector ISAs, and requires a specific JVM. Both approaches abstract SIMD instructions, which limits the ability of a developer to tune the code to a particular microarchitecture.

Autovectorization in the JVM. Initially introduced in the Jikes RVM [4], the HotSpot JVM uses SLP [11] based autovectorization. SLP is limited and is only able to vectorize basic blocks consisting of groups of isomorphic instructions, generating SSE and AVX code. Partial support of FMA and AVX-512 is only planned for Java 9 [9].

Figure 7. Performance analysis: Variable Precision.
Support of low-level code in the JVM. Sulong [18] is a system to execute low-level languages such as C/C++, Fortran, Ada, and Haskell in the JVM. Sulong is capable of handling low-level languages that compile to LLVM, using LLVM IR interpreter built on top of the Truffle framework [31], running on the Graal VM. While this approach can bring a support of low-level instructions in the JVM, it does not support SIMD instructions, as Graal does not provide sufficient analysis for vectorization. Furthermore, due to interpretation, Sulong is shown to be outperformed by native compilers such as gcc.

Automatic generation of DSLs in LMS. DSLs have been generated into LMS before. Yin-Yang [10] automatically generates deep DSL embeddings from their shallow counterparts by reusing the core translation. Forge [28] generates DSLs from a declarative specification. None of the approaches have been challenged to generate DSLs of the scale imposed by the large amount of SIMD intrinsics, nor were they designed to automatically infer effects of mutability.

SIMD intrinsics in LMS. A limited support of SIMD instructions has been introduced while abstracting vectors architectures [27]. This approach has been used in generating libraries for high-performance code, and integration with the JVM has not been demonstrated. On an even lower level, LMS has been used to define domain-specific ISAs by generate specialized hardware [6, 7].

6 Conclusion

Our work shows how metaprogramming techniques can be used to bridge the gap between high-level managed languages and the need to access low-level instructions in high performance code development. Specifically, we showed how to provide access to SIMD intrinsics in the HotSpot JVM, thus eliminating the need to write C/C++ code. Two key techniques underlie our approach. First is the use of embedded DSLs to express intrinsics inside JVM languages such as Scala. These are generated directly from the vendor XML specification, which enables complete intrinsics support and fast updates in the future. Second is the use of staging to convert SIMD intrinsics interspersed with Scala code into high-performant C kernels, which are then compiled and linked via JNI. The challenge in our work is in the systematic handling of large sets of functions, converting them into sets of DSLs, automatically inferring their side effects, and creating a compiler and code generation pipeline for convenient and productive development. We show how the SIMD support in JVM can be used to build powerful high-level and low-level abstractions while offering significant, often manifold speedup over the autovectorizing HotSpot JIT compiler.

A Artifact Description

Submission and reviewing guidelines and methodology:
http://cTuning.org/ae/submission-20160509.html

A.1 Abstract

To reproduce the results presented in our work, we provide an artifact that consist of two parts:

- lms-intrinsics, a precompiled jar library that includes all Intel-based SIMD intrinsics functions, implemented as Scala eDSLs in LMS.
- NGen, a runtime implemented in Scala and Java, that enables the use of lms-intrinsics in the JVM and includes the experiments discussed in our work.

The SIMD based eDSLs follow the modular design of the LMS framework and are implemented as an external LMS library, separated from the JVM runtime. This allows the standalone use of lms-intrinsics, enabling LMS to generate x86-vectorized code outside the context of the JVM. The JVM runtime (NGen) demonstrates the use of the lms-intrinsics by providing the compiler pipeline to generate, compile, link, and execute the LMS-generated SIMD code and has a strong dependency on this library.

The experiments included in the artifact come in the form of microbenchmarks. While the most convenient deployment for this artifact would have been a Docker image through Collective Knowledge, we decided to eliminate the overhead imposed by the containers and provided a bare metal deployment that aims at providing as precise results as possible for our tests. To achieve that, we use the SBT (Simple Build Tool) to build and execute our experiments.

A.2 Description

A.2.1 Check-List (Artifact Meta Information)

- **Algorithm:** Using SIMD intrinsics in the JVM. Experiments include dot-product on quantized arrays, BLAS routines: SAXPY and Matrix-Matrix-Multiplication.
- **Compilation:** lms-intrinsics is a precompiled library, compiled with Scala 2.11 and is available as a jar bundle, accessible through Maven. NGen requires Scala 2.11 and Java 1.8 for compilation. Both NGen and lms-intrinsics generate C code that is compiled with GCC, ICC, or LLVM.
- **Transformations:** To make SIMD instructions available in the JVM, NGen uses LMS as a staging framework. The user writes vectorized code in the eDSL in Scala, and NGen stages the code through multiple compile phases before execution.
- **Binary:** lms-intrinsics is a jar bundle. NGen includes binaries for SBT v0.13.6, as well as a small library for CPUID inspection and Sigar v1.6.5_01 (System Information Gatherer And Reporter https://github.com/hyperic/sigar) binaries. NGen has various dependencies on precompiled libraries that include Bridj, Apache Commons, ScalaMeter, Scala Virtualized, LMS, and lms-intrinsics. SBT automatically pulls all dependencies and their corresponding versions.
- **Data set:** Our experiments operate with random data, requiring no data set.
A.2.3 Hardware Dependencies

lms-intrinsics as well as NGen are able to generate C code that can run on any x86 and x86-64 architecture supporting Intel ISAs. However, the full set of our experiments requires at least a Haswell machine. Namely:

- **SAXPY and MMM algorithms** are implemented using AVX and FMA ISAs, and therefore require at least a Haswell enabled process. Broadwell, Skylake, Kaby Lake or later would also work.
- The dot product of the quantized arrays relies on AVX2, and FMA flags, but also uses the hardware random number generator, requiring the RDRAND ISA, as well FP16C to deal with half-precision floats.

We recommend disabling Intel Turbo Boost and Hyper-Threading technologies to avoid the effects of frequency scaling and resource sharing on the measurements. Note that these technologies can be easily disabled in the BIOS settings of the machines that have BIOS firmware. Many Apple-based machines, such as the MacBook or others, do not have a user accessible BIOS firmware, and could only disable Turbo Boost using external kernel modules such as Turbo Boost Switcher (https://github.com/rugarciap/Turbo-Boost-Switcher).

A.2.4 Software Dependencies

lms-intrinsics is a self-contained precompiled library and all of its software dependencies are handled automatically through Maven tools such as SBT. To build and run NGen, the following dependencies must be met:

- Git client, used by SBT to resolve dependencies.
- Java Development Kit (JDK) 1.8 or later.
- C compiler such as GCC, ICC or LLVM.

After installing the dependencies, it is quite important to have the binary executables available in the $PATH. This way, the SBT tool will be able to process all compilation phases as well as to execute the experiments. Make sure that the following commands work on your terminal:

```
> git --version
gcc --version
java --version
javac --version
```

It is also important to ensure that the installed JVM has an architecture that GCC can compile to. This is particularly important for Windows users: the 32-bit MinGW port of GCC will fail to compile code for 64-bit JVM.

A.3 Installation

The artifact can be cloned from the GitHub repository:

```
git clone https://github.com/astojanov/NGen
```

The artifact already includes a precompiled version of SBT. Therefore, to start the SBT console, we run:

```
cd NGen
# For Unix users:
./bin/sbt/bin/sbt
# For Windows users
bin\sbt\bin\sbt.bat
```

Once started, we can compile the code using:

```
> compile
```

Once invoked, SBT will automatically pull lms-intrinsics as well as all other dependencies and start the compilation.

A.4 Experiment Workflow

Once SBT compiles the code, we can proceed with evaluating our experiments. We do this through the SBT console. To inspect the testing machine through the NGen runtime we use:

```
> test-only cgo.TestPlatform
The runtime will be able to inspect the CPU, identify available ISAs and compilers and inspect the current JDK. If the test platform is successfully identified, we can continue with the experiments.

**Generating SIMD eDSLs.** The lms-intrinsics bundle includes the automatic generator of SIMD eDSLs, invoked by:

```
> test-only cgo.GenerateIntrinsics
```

The Scala eDSLs (coupled with statistics) will be generated in the Generated_SIMD_Intrinsics folder.

**Explicit vectorization in the JVM.** To run the experiments depicted in our work, we use:

```
> test-only cgo.TestSaxpy
> test-only cgo.TestMMM
> test-only cgo.TestPrecision
```

In the case of SAXPY, if the testing machine is not Haswell based, we provided an architecture-independent implementation of SAXPY:

```
> test-only cgo.TestMultiSaxpy
```

Each result shows the size of our microbenchmarks, and the obtained performance in flops/cycle.

### A.5 Evaluation and Expected Result

In the evaluation of the experiment workflow, we expect LMS to produce correct vectorized code using lms-intrinsics. Furthermore, we expect the performance results to behave consistent with the results shown in this paper, outperforming the JVM on the microarchitectures that support our experiments. Finally, we expect the automatic generation of eDSLs to be easily adjustable to subsequent updates on the Intel Intrinsics specifications.

### A.6 Experiment Customization

There are many opportunities for customization. We can use NGen to easily develop vectorized code, and we can use ScalaMeter to adjust the current benchmarks.

**Developing SIMD code.** The NSaxpy.scala class, available in src/ch/ethz/acl/ngen/saxpy/, provides detailed guidelines for the usage of SIMD in Scala. Following the comments in the file, as well as the structural flow of the program, one can easily modify the skeleton to perform other vectorized computations.

**Customizing Benchmarks.** Each performance experiment, uses ScalaMeter and is implemented as a Scala class. The Matrix-Matrix-Multiplication includes BenchMMM.scala located in src/ch/ethz/acl/ngen/mmm/. The implementation allows changes to various aspects of the benchmarks, including the size and the values of the input data, warm up times, different JVM invocations, etc.

### Acknowledgments

This research was partially supported under NSF awards 1553471 and 1564207, and DOE award DE-SC0018050. The authors also wish to thank Kevin J. Brown for the inspiring discussion that lead to the creation of our low precision ISA.

### References


