VOBLA: A Vehicle for Optimized Basic Linear Algebra

Ulysse Beaugnon[1,2] Alexey Kravets[1] Sven van Haastregt[1] Riyadh Baghdadi[2] David Tweed[1]
Javed Absar[1] Anton Lokhmotov[1]

[1]ARM, United Kingdom
firstname.lastname@arm.com

[2]INRIA and École Normale Supérieure, France
firstname.lastname@inria.fr

Abstract

We present VOBLA, a domain-specific language designed for programming linear algebra libraries. VOBLA is compiled to PENCIL, a domain independent intermediate language designed for efficient mapping to accelerator architectures such as GPGPUs. PENCIL is compiled to efficient, platform-specific OpenCL code using techniques based on the polyhedral model. This approach addresses both the programmer productivity and performance portability concerns associated with accelerator programming.

We demonstrate our approach by using VOBLA to implement a BLAS library. We have evaluated the performance of OpenCL code generated using our compilation flow on ARM Mali, AMD Radeon, and AMD Opteron platforms. The generated code is currently on average $1.9\times$ slower than highly hand-optimized OpenCL code, but on average $8.1\times$ faster than straightforward OpenCL code. Given that the VOBLA coding takes significantly less effort compared to hand-optimizing OpenCL code, we believe our approach leads to improved productivity and performance portability.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent programming

General Terms Algorithms, languages, performance

Keywords linear algebra; GPU; domain-specific language; parallel; sparse matrix; BLAS

1. Introduction

Programming accelerators such as (GP)GPUs is accomplished today using rather low-level Application Programming Interfaces (APIs) such as OpenCL, which raises concerns from the programmer productivity and performance portability perspectives. *Programmer productivity* is affected because low-level APIs distract the programmer from the actual problem. *Performance portability* is affected because code optimized for a particular accelerator often underperforms on a different accelerator. We present the compilation flow depicted in Figure 1 that aims to address both concerns for the domain of linear algebra. This compilation flow includes VOBLA, a novel Domain-Specific Language (DSL) for linear algebra, which is the key focus of this paper.

LCTES '14, June 12–13 2014, Edinburgh, United Kingdom.

ACM 978-1-4503-2877-7/14/06. . . $15.00.
http://dx.doi.org/10.1145/2597809.2597818

VOBLA provides basic operations and traversals of array types while hiding implementation and layout details. This allows a programmer to describe an operation at the domain level in an intuitive way, avoiding to deal with low-level memory allocation and layout. The result is compact, readable, and maintainable code. This addresses the concern of programmer productivity.

VOBLA is compiled into PENCIL, a C99-based platform-neutral compute intermediate language [1], while retaining sufficient information for generating efficient accelerator code. PENCIL is then compiled into OpenCL code optimized for a specific accelerator using polyhedral compilation techniques that make use of the retained information. This addresses the concern of performance portability.

The main contributions of our paper can be summarized as follows:

- VOBLA: A DSL that can compactly represent linear algebra operations, separating functional semantics from implementation details such as storage layouts. Any parallelism inherent in the function is not obscured by implementation details, enabling parallel code generation.
- A Basic Linear Algebra Subprograms (BLAS) implementation in VOBLA to prove the advantages of VOBLA and of our compilation flow.
- A compilation flow that links domain-specific languages such as VOBLA to the performance-portable intermediate language PENCIL, and the compilation of PENCIL to optimized OpenCL. The VOBLA and PENCIL tools are available as open-source software [8].

The rest of this paper is organized as follows. Section 2 outlines our solution approach. Section 3 motivates our choice for a new DSL and compares other approaches to obtain linear algebra libraries. Section 4 introduces PENCIL and Section 5 introduces VOBLA. Section 6 describes how user-defined matrix layouts are described and handled in VOBLA. Section 7 describes a VOBLA implementation of a BLAS library, and Section 8 reports performance results of VOBLA BLAS routines.

2. Solution Approach

The compilation flow in Figure 1 depicts our solution to the problems of programmer productivity and performance portability. Our flow starts with the domain of linear algebra, covering a wide range of operations on vectors and matrices. While one could program linear algebra solutions directly in OpenCL, it would be cumbersome with no guarantee of good performance across different GPUs. Depending on which platform the code is written for and which platform the code is running on, the performance will vary. So even though direct OpenCL programming gives some degree of functional portability, performance portability is lacking.

The last step of our compilation flow translates PENCIL code to OpenCL code. In this step we use the Polyhedral Parallel Code

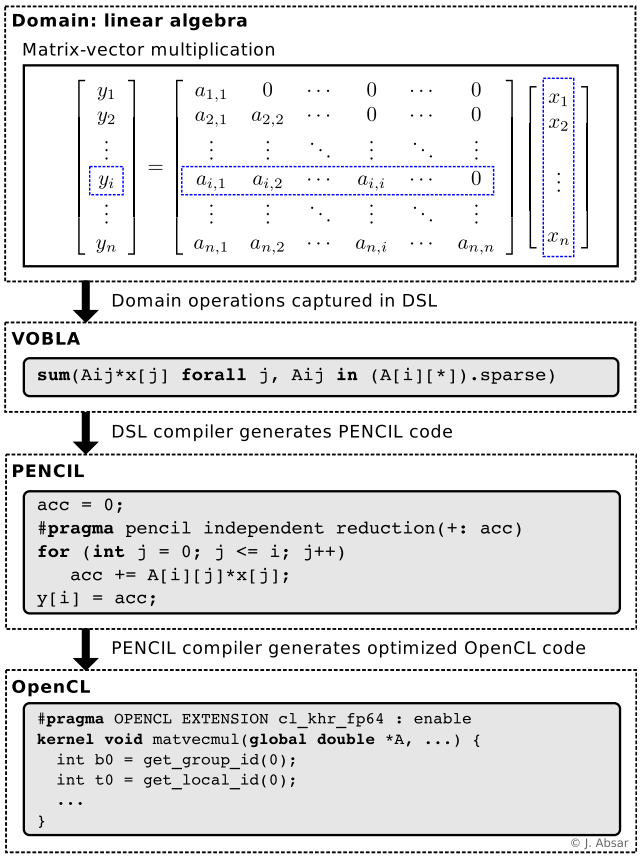

Figure 1. Compilation Flow.

Generator (PPCG) [16] which is a tool that employs polyhedral compilation techniques to generate parallel GPU code. Polyhedral compilation techniques are effective in improving parallelism and locality for codes that involve array accesses inside loop nests where the access functions for the array indices are affine combinations of loop iterators. Not all codes can be optimized easily by polyhedral compilation techniques. For example, irregular codes such as tree and graph traversals are more difficult to analyze. Our compilation flow is specially suited for the domains that we call *polyhedral-friendly* such as linear algebra, scientific computation, and image processing domains.

3. Related Work

In this section, we review work on domain-specific language development and work on implementation of linear algebra libraries. In particular, we focus on linear algebra libraries for GPUs.

3.1 Domain-Specific Languages

Domain-specific languages have existed for a long time. The programmable looms and the playable piano scrolls are probably the earliest examples of domain-specific languages. Even some of the early general programming languages were essentially domain-specific such as COBOL (COmmon Business-Oriented Language) and Fortran (FORmula TRANslation) as they were designed with a specific class of applications in mind. Then there was an era of general-purpose programming languages (1970-2000) which saw many flavors – imperative, declarative, object-oriented, logic and functional – and combinations of them develop and flourish.

In the last decade, there has been a renaissance in DSLs. Van Deursen et al. [15] give a survey of the literature on domain-specific languages. They list 75 key publications, at the time of their publication in 2000, with a brief description for each. While van Deursen et al. list best practices in DSL design, Kelly et al. [7] present "what NOT to do", based on an analysis of 76 Domain-Specific Modeling (DSM) languages spanning 15 years, four continents, and around 100 language creators. They present worst practices in DSL development, in the order DSL developers would encounter them over the life of their project. Some of the interesting ones they list are:

- Poor domain understanding – Insufficiently understanding the problem domain (17%) or the solution domain (5%).
- Sacred at birth – Viewing the initial language version as unalterable (12%).
- Theoretically great – Wanting the language to be theoretically perfect, leaving implementation as secondary concerns (8%).
- Poor use case – Ignoring largely the language's usage (42%).

We do not claim that we have avoided all mentioned pitfalls. But we have tried to avoid many by coding a key domain application, that is, a BLAS library, in our VOBLA DSL.

Many papers and books classify DSLs into external and internal [3]. An *external* DSL is a domain-specific language that has a syntax different from the programming language implementing the DSL. For example, a MATLAB compiler may be written in C; the language syntax and semantics of MATLAB are quite different from C's syntax and semantics. An *internal* DSL is a language represented within the syntax of a general purpose language. Powerful functional languages like Haskell and Scala provide stylized use of the language to present the look and feel of a separate DSL to a domain programmer. Internal DSL are referred in most literature as *embedded* DSLs, but we, like Fowler et al. [3], prefer the term internal since embedded has an unintended connotation of embedded systems. Typically, an external DSL requires more work to build compared to an internal DSL. For an external DSL, one needs to define the grammar and implement the entire compiler framework.

Delite [5] uses a hybrid approach of internal and external DSLs. They use the concept of *language virtualization* to characterize a host language (Scala) that lets them implement embedded DSLs that are virtually indistinguishable from stand-alone DSL. The virtualized host language provides the front-end that the DSL can use, while letting the DSL leverage meta-programming facilities to build and optimize an IR. However, the Delite approach does not leverage the polyhedral framework and we are looking to see if Delite output could be channeled to generate PENCIL code which could then leverage our compilation framework.

The Halide DSL language and compiler [11] targets image processing algorithms. Halide uses an interval analysis technique which, as the authors put it, *"is simpler and less expressive than polyhedral model, but more general in the class of expressions it can analyze"*.

3.2 Linear Algebra Libraries

In addition to developing VOBLA, we have used VOBLA to implement BLAS. Due to its extensive use in industry and academia, many tuned implementations of BLAS have been developed in the past. Some examples of industry tuned BLAS are Apple's Accelerate framework for MAC OS X and iOS; AMD's Accelerated Parallel Processing Math Libraries (APPML); Microsoft's C++ AMP BLAS [2], leveraging C++ AMP language for GPU programming; and Intel's Math Kernel Library (MKL).

The main difference between these implementations and our implementation is that our implementation has been coded in a high-

level DSL, whereas the other implementations have been coded and hand-optimized and so are not performance portable. Other notable examples of implementations of BLAS include Automatically Tuned Linear Algebra Software (ATLAS) [18], which provides APIs in C and Fortran; Kazushige Goto's BLAS [6] which is tuned for x86 and AMD64 processor architectures by means of handcrafted assembly code; and GNU Scientific Library [4], which provides generic BLAS implementation for GNU/Linux.

OoLaLa (Object Oriented anaLysis And design of numerical Linear Algebra) [10] models matrices in terms of properties and storage formats similar to VOBLA, e.g. using iterators to scan sparse matrices. Since it is Java based, the linear algebra function implementations are not as compact as the same coded in VOBLA. But the key difference is that the VOBLA compiler generates PENCIL which is passed through polyhedral optimization to generate parallel OpenCL code, while OoLaLa executes the code in a Java virtual machine, which makes it portable but not performance-oriented.

The Vienna Computing Library [14] addresses specifically the performance and portability of linear algebra codes for GPUs. It proposes device-specific OpenCL code generation at runtime, through use of common code templates and supplemented by an auto-tuning framework for portable performance. Many of the elements of ViennaCL, such as auto-tuning, can be integrated into our approach for efficient OpenCL code generation, while still taking advantages of programmer productivity of VOBLA.

uBLAS [17] is another high-level portable BLAS library. It is a C++ template class library and provides BLAS level 1, 2, 3 functionality for dense, packed, and sparse matrices.

4. PENCIL

PENCIL (Platform-Neutral Compute Intermediate Language) is an intermediate language for accelerator programming [1]. It is a C99-based language that uses a strict subset of C constructs with additional annotations provided by the programmer or a DSL compiler to capture meta information about the program. The following are three key characteristics of PENCIL.

First, the language itself does not specify which parts of the program must be executed on an accelerator device. Such decisions are intended to be made by device-specific compilers instead. Second, every PENCIL program can be considered as a semantically equivalent sequential C program, by ignoring the annotations. This helps for example in debugging of PENCIL programs. Third, all meta information in PENCIL is optional.

Any C program that uses only the allowed subset of C constructs is a valid PENCIL program with the same semantics.

4.1 Restrictions

Pointers, jumps, global non-constant variables, unions, and bitfields are forbidden in PENCIL. These restrictions make the code analyzable by polyhedral compilation techniques.

PENCIL also imposes some restrictions on the allowed C constructs:

- *A* **for***-loop must be a counted loop with a constant stride:*

```
for (int iter = 0; iter < N; iter += step) {
  //Body
}
```

- *A compound assignment cannot be part of an expression:*

```
int i = 2;
int j = i += 2; //Forbidden in PENCIL.
```

  This eliminates constructs with undefined behaviour such as:

```
int j = i++ + ++i;
```

  This restriction enables PENCIL compilers to apply loop transformations using polyhedral compilation techniques.

- *An array argument must be declared using the C99 variable-length array syntax and using the* **restrict**, **const**, *and* **static** *type qualifiers:*

```
void foo(int n, int m,
  float a[static const restrict n * m]) {}
```

  This restriction simplifies memory access domain analysis, since the shape of the array is known at compile time.

- *Calling external C functions is forbidden.*

These restrictions make it possible to represent PENCIL in the polyhedral model, while keeping PENCIL expressive enough to represent irregular algorithms. For irregular code, whose properties cannot be directly extracted from the implementation code, PENCIL meta information should be used to provide such information.

4.2 PENCIL Meta Information

PENCIL provides various ways for a DSL compiler or programmer to provide program meta information that cannot be captured in regular program code.

Loop annotations

A PENCIL loop can be annotated with a pragma to mark high-level loop properties:

- Independent pragma: this pragma specifies that there are no dependences between iterations of a loop:

```
#pragma pencil independent
for (i = 0; i < n; i++) {
  B[T[i]] = foo(i);
}
```

- Reduction pragma: extends the independent pragma to specify that a loop does not contain loop-carried dependences except for dependences through a reduction variable:

```
#pragma pencil independent reduction(+: result)
for (i = 0; i < n; i++) {
  B[T[i]] = foo(i);
  result += A[i];
}
```

Inline memory access information

Additional memory access information can be provided for every PENCIL statement to specify memory access information that cannot be derived from the code.

```
#pragma pencil access\
  {USE(A); USE(B); DEF(A);}
for(int i = 0; i < N; i++) {
  A[B[i]] *= 2;
}
```

`USE` and `DEF` are polymorphic built-in functions and can be used to mark the usage (read access) or definition (write access) of individual locations, that is, variables or elements of an array.

Summary functions

Alternatively, additional memory access information can be provided in the form of a summary function:

```
void
summary1(int N, int A[const static restrict N],
         int B[const static restrict N]) {
  USE(A);USE(B);DEF(A);
```

```
}
#pragma pencil access {summary1(N, A, B);}
for(int i = 0; i < N; i++) {
  A[B[i]] *= 2;
}
```

Built-in functions

PENCIL provides a set of built-in functions which capture program meta information:

- **void** `ASSUME(cond)`: specifies that a PENCIL compiler can assume that the expression `cond` is true at the given program point. `ASSUME` proves to be useful in situation where otherwise conservative conclusion by the compiler would lead to insufficient optimization. For example, using the assume information below, a compiler can infer that there is no dependence present between the read and write to the same array.

```
ASSUME(m > n)
for (i = 0; i < n; i++) {
  D[i] = D[i+m];
}
```

- **void** `KILL(loc)`: marks its argument as dead at this point, that is, it is not used after this point.

5. VOBLA

VOBLA is designed for implementing linear algebra libraries like BLAS [12] or LAPACK [13]. Unlike some other DSLs like MATLAB, VOBLA only provides a small set of built-in functions. It focuses on providing a generic and compact representation of basic linear algebra algorithms that can be compiled into highly efficient code for GPUs, and allowing to use the accelerated code in existing software.

Compactness is achieved using template functions that are instantiated for different data types, floating-point precisions, and array storage formats. Ease of programming is achieved by using a restricted set of vector operators and iterators over arrays. Code performance is achieved by annotating the PENCIL code with information essential for generating efficient GPU code. In addition, we ensure that access patterns remain compatible with the polyhedral model such that advanced compilation techniques can be used to obtain efficient GPU code. Compatibility with existing software is enabled both by allowing generic functions that can be adapted to existing data formats and by ensuring existing libraries can be expressed in VOBLA.

5.1 General Description

VOBLA functions are defined using an imperative programming style to match the semantics of existing libraries written in C or Fortran. VOBLA provides three control flow operators: **`for`**, **`if`**, and **`while`**. The **`if`** and **`while`** operators have semantics similar to their counterparts in the C and PENCIL languages. The **`for`** operator has the following three key features.

First, the programmer can specify that iterations of a loop are independent of each other by using the **`forall`** keyword instead of **`for`**. A loop without **`forall`** can still be parallelized by a downstream PENCIL compiler but this requires more analysis by the downstream compiler.

Second, a **`for`** or **`forall`** loop specifies a multi-dimensional iteration space instead of a single-dimensional iteration space. A multi-dimensional iteration space is specified by a comma-separated list of single-dimensional iteration spaces. A single-dimensional iteration space specifies a scalar range or array to iterate over. A scalar range can be of the form $\langle lb : ub \rangle$, where lb and ub are the inclusive lower and upper bounds. Iterating over dense or sparse arrays will be explained in Section 5.4. The following loop iterates over a 3-dimensional space, with each iteration being independent of the other iterations.

```
forall(i in 0:n-1, j in 0:i-1, k in j:i-j) {
  <body>
}
```

Third, VOBLA provides a compact way to express loops over assignments. The following statement assigns i to X_i for $0 \leq i < n$:

```
X[i] = i forall i in 0:n-1;
```

⟨*assigment*⟩	::=	⟨*expr.*⟩ ⟨*assign*⟩ ⟨*expr.*⟩
⟨*assign*⟩	::=	'=' | '+=' | '−=' | '⋆=' | '/='
⟨*op*⟩	::=	'+' | '−' | '⋆' | '/'
⟨*len*⟩	::=	`len` '(' ⟨*expr.*⟩ ')'
	|	`len` '(' ⟨*expr.*⟩ ',' ⟨*int*⟩ ')'
⟨*expr.*⟩	::=	⟨*expr.*⟩ ⟨*op*⟩ ⟨*expr.*⟩
	|	'−' ⟨*expr.*⟩
	|	'(' ⟨*expr.*⟩ ')'
	|	`sum` '(' ⟨*expr.*⟩ `forall` ⟨*iter. space*⟩ ')'
	|	⟨*function call*⟩
	|	⟨*array access*⟩
	|	⟨*len*⟩
	|	⟨*builtin call*⟩

Figure 2. Partial grammar of VOBLA expressions

5.2 Expressions

A partial grammar of VOBLA expressions is shown in Figure 2. The basic operators are also available for vector and vector-scalar operands, as we will describe in Section 5.5. The **`len`** operator returns the size of the different dimensions of an array. The **`sum`** operator provides a compact notation to sum a sequence of scalars, an operation that is often needed in the linear algebra application domain. As an example of the **`sum`** operator, the following statement computes $\sum_{x \in X} x \cdot x$ and assigns it to `norm2`:

```
let norm2 = sum(x*x forall _, x in X);
```

Given that X is a one-dimensional array, the **`forall`** keyword is followed by a comma-separated pair of the iterator and a variable holding the values of X. We do not use the iterator value in the above computation, so we discard the iterator using the _ symbol.

VOBLA provides the **`Conjugate`**, **`Re`** (real part) and **`Im`** (imaginary part) built-in functions, as well as built-in functions for real and integer scalars such as `sqrt` that have direct OpenCL counterparts.

5.3 Template Functions

Linear algebra libraries often implement many versions of the same function that only differ in the data type, the floating-point precision, or the way arrays are stored. VOBLA addresses this characteristic using template functions. Inside template functions, only generic types are used. The VOBLA compiler generates different versions of the template function with different types. Template functions improve the ease of programming and compactness of the code by hiding details and exposing only the algorithm. Template functions also help the compiler since they allow better analysis of the function as the high-level structure of the algorithm is not obfuscated by implementation details.

Functions are either automatically instantiated when called from another VOBLA function or explicitly instantiated with an **export** statement as follows:

```
function scal(a: Value, out X: Value[]) { ... }

export scal<Complex Double>(X is Column) as zscal;

export scal<Float>(X is Reversed Column) as sscal;
```

The first export statement exports a version of `scal` that operates on double-precision complex numbers and treats `X` as a plain array. The second export statement exports a version of `scal` that operates on single-precision real numbers and treats `X` as a reversed array.

5.3.1 Scalar Types

VOBLA provides a generic scalar type **Value** that is specialized to **Real** or **Complex** during template function instantiation. The **Real** and **Complex** types are still generic, as these can be specialized into single-precision (**Float** and **Complex Float**) and double-precision (**Double** and **Complex Double**) types. The specialization of **Value** operates at the function level such that **Value** can be treated as a standard type inside a function. VOBLA provides the **Index** type for integer types.

5.4 Array Access Patterns

To hide the complexity of storage formats, provide genericity, and simplify code generation, arrays can only be accessed through access patterns. VOBLA provides three built-in access patterns:

- **Iterate**: This access pattern enumerates the elements of an array in an undefined order. For example, to compute

  $$A_{i,j} \leftarrow i \cdot j \quad \forall\, 0 \leq i < m,\ 0 \leq j < n$$

  for an $m \times n$ matrix A (where A can be a sparse matrix, meaning elements which are zero are not stored) using the Iterate access pattern, one writes the following VOBLA code:

  ```
  Aij = i * j forall i, j, Aij in A;
  ```

  The Iterate access pattern is useful for repeating an operation on each element of an array. The Iterate access pattern should be used when it is impossible to access a specific element efficiently from its indices, which is the case for most sparse matrix formats.

- **Sparse Iterate**: This access pattern operates similar to Iterate, but skips the elements of the array that are known to be zero at compile time. For example, for a lower-triangular matrix, we know at compile time which elements are zero. Some zero elements might still be enumerated if their locations are not known at compile time. Thus, this access pattern can only be used if the zero elements have no effect on the result. The following VOBLA code computes $\sum x \cdot x$ for all elements $x \in X$ known to be non-zero at compile-time:

  ```
  let norm2 = 0;
  norm2 += Xi * Xi for _, Xi in X.sparse;
  ```

- **Indexed**: This access pattern allows accessing an element from its indices. The Indexed access pattern is more expressive than the Iterate and Sparse Iterate access patterns, but it cannot be implemented for every storage format. For example, it is impossible to efficiently access an element given its coordinates for most of the sparse matrix formats. This access pattern is necessary when multiple arrays are involved, as it is impossible to iterate over multiple arrays at once. The following VOBLA code accesses a 1-dimensional array `X` using the Sparse Iterate access pattern, which yields the index `i`. This index is then used to access an array `y` using the Indexed access pattern.

  ```
  let dot = 0;
  // y is accessed using Indexed access
  dot += xi * y[i] forall i, xi in X.sparse;
  ```

The Sparse Iterate access pattern can be implemented using the Iterate access pattern by not skipping any zero elements. The Iterate access pattern can be implemented using the Indexed access pattern by accessing all the elements one-by-one. Thus, any array implementing Iterate also implements Sparse Iterate and any array implementing Indexed implements both Iterate and Sparse Iterate.

We believe these three access patterns are expressive enough to cover a wide range of storage formats in linear algebra. This assumption has been confirmed by the implementation of BLAS described in Section 7. Custom access patterns can be defined as well, for example to iterate over the rows of a matrix instead of iterating over the scalar elements. Defining custom access patterns is discussed in Section 6. A current limitation of VOBLA is that custom access patterns cannot be used with built-in operators.

5.5 Array Operators

VOBLA allows a compact representation of basic array operations such as element-wise scaling. They make use of the different access patterns described in Section 5.4. The VOBLA compiler translates array operators into equivalent scalar code using iterators. The scalarized code iterates over one array and uses the Indexed access pattern on the other array. Therefore, one of the operands must implement the Indexed access pattern.

As an example, consider the following VOBLA code that computes

$$X_i \leftarrow X_i + 2Y_i, \quad \forall\, 0 \leq i < n,$$

where n equals the number of elements in X and Y:

```
X += 2*Y;
```

The assignment to `X` is scalarized by accessing `X` using the Indexed access pattern and accessing `Y` using the Sparse Iterate access pattern. This yields the following equivalent VOBLA code:

```
X[i] += 2*Yi forall i, Yi in Y.sparse;
```

Operations on two arrays require the arrays to have the same dimensionality and size. When possible, our VOBLA compiler uses the Sparse Iterate access patterns for array operations such that zero-elements are skipped. When the compiler has to choose on which array to iterate, it currently picks an arbitrary one. As a future optimization, the compiler could choose the array with the least number of non-zero elements to apply Sparse Iterate to.

5.6 Array Views

Certain linear algebra operations such as matrix transposition can be performed by only changing the way an array is accessed, avoiding expensive copy operations. Since the algorithm is decoupled from the way the array is stored in VOBLA, we leverage this for the following operations:

- **Conjugate**, **Re** (real part), and **Im** (imaginary part) for arrays.
- **Transpose**, **Diagonal**, and **AntiDiagonal** for dense matrices.
- Extraction of array substructures for dense arrays, such as:

  ```
  X[2:4]  // Take elements of X between 2 and 4
  A[*][2] // Take the third column of A
  ```

For example, the following transposition of matrix `A` in VOBLA

```
let a = Transpose(A)[i][j];
```

is translated into the following PENCIL code:

```
float a = A[j][i];
```

The operations listed above can also be applied to function arguments. Instead of applying the operation before calling the function, a new version of the called function is generated that accesses the data according to the operation. This helps reducing the number of unnecessary array copy operations at the expense of a larger code size. Arrays are only copied when it is specified explicitly using the assignment operator:

```
B = Transpose(A);     // A is copied
foo(Transpose(A));    // A is not copied
```

6. User Defined Access Patterns

By default, VOBLA provides the Iterate, Sparse Iterate, and Indexed access patterns defined in Section 5.4. VOBLA allows defining new access patterns and storage formats to cope with user defined access patterns and storage layouts. We use four different objects to define new access patterns. They are defined below and explained in detail in the remainder of this section.

- **Interface**: An Interface object defines the type representing an array when declaring a function. It specifies which access pattern is implemented by the array.
- **Storage**: A Storage object contains the storage layout for an array. Storage objects do not contain information on how to access data.
- **Layout**: A Layout object describes how to access data for a given storage format. A Layout object implements an interface.
- **View**: A View object describes layouts in terms of existing ones using inheritance, enabling reuse of layout definitions.

6.1 Interfaces

Interface objects allow a VOBLA programmer to declare arrays in function headers. They represent the different ways an array can be used. Interface objects only exist to implement the type system and do not contain any information on how the array data is accessed. During compilation, each array will have a layout implementing its interface specifying how to access the array data.

Interface objects can be considered as an object-oriented feature. They declare the access methods that the Interface provides. Only the prototypes of Interface methods are defined in an Interface object; the method implementations are defined in Layout objects. Interface objects can inherit from other Interface objects.

6.1.1 Interface Methods

To access an array, a method defined by an Interface object is called. Unlike real object-oriented languages, the methods of an Interface are only used to access the array data. They are not meant to perform computations and cannot have side effects. The built-in access patterns defined in Section 5.4 are also defined using Interface objects. Array accesses using a particular interface are made through the methods described in Table 1.

Access Pattern	Method	Interface
Iterate	`iterate`	`Iterable`
Sparse Iterate	`sparse`	`SparseIterable`
Indexed	`[<Index>]`	`Accessible`

Table 1. Built-in access patterns, methods, and interfaces.

Methods can return a reference to an element of the array. This is the only way to modify an array. References are denoted by `&` and can only be used in method return types. The following VOBLA code declares a method prototype named `access` for accessing a 1-dimensional array:

```
access(i: Index): &Value;
```

Methods can return a range of elements. In such a case, the method can only be used as an iterator in a **`for`** or **`forall`**. Ranges may contain references such that all elements in the range can be modified. The following VOBLA code declares a method prototype named `iterate` that returns a range of references:

```
iterate(): range<Index, &Value>;
```

Methods can also return an Interface object representing a different view of the same data. No data is copied in such a case, only the way the data is accessed is changed. The following VOBLA code declares a method prototype named `re` that returns the `Accessible` interface specialized for **`Real`** types:

```
re(): Accessible<Real>[];
```

6.1.2 Interface Specialization

Interfaces are not used directly for array type declarations. They are first optionally specialized by a scalar type and an optional read-only qualifier. The scalar specialization replaces **`Value`** by a type such as **`Complex`** or **`Real`**, or retains **`Value`**. The read-only qualifier **`in`** replaces any reference types by their equivalent non-reference types and makes returned views read-only. The following function argument declaration declares argument `A` as a read-only argument inheriting the SparseIterable interface on the generic **`Value`** type:

```
in A: SparseIterable<Value>[m][n]
```

This means **`Value`** can be specialized when exporting the declared function.

The read-only qualifier enables optimizations in the downstream PENCIL compilation process by flagging PENCIL arrays as **`const`**. A non-read-only interface can always be casted to a read-only interface. Read-only interfaces can also be casted to the same interface with a weaker scalar type specialization. For example, a constant array of **`Real`** can be casted to a constant array of **`Complex`** with the same interface.

6.1.3 Interface Inheritance

When defining a new interface, one or more interfaces can be specified as a base interface. The defined interface then inherits the access patterns of the base interfaces. The following VOBLA code defines a new interface for triangular matrices. The `triangle` method returns the non-zero triangular part of the matrix. The `diagonal` method returns the diagonal of the matrix.

```
interface TriangularMatrix[][]
          implements SparseArray<Value>[][] {

  triangle(): Triangle<Value>[][];
  diagonal(): Array<Value>[];
}
```

Every type implementing this interface must implement the methods of the base interface `SparseArray`, and the `triangle` and `diagonal` methods specified in this `TriangularMatrix` interface.

Types implementing the derived interfaces also implement all base interfaces. Thus, any object which type implements the `TriangularMatrix` interface can be passed to a function expecting a `SparseArray<Value>` interface.

Inherited interfaces must be specialized. This restricts the derived interface to only implement the specific version of the base inter-

face. For example, the interface for complex sparse matrices can be defined as follows:

```
interface ComplexSparse[][]
          implements SparseArray<Complex>[][] {
}
```

6.2 Storages

The data storage layout of an array is described using a storage object. A storage object does not describe how the data is accessed and the data in the same storage may be accessed in different ways. Thus, a storage object represents the part of an array that is independent of the way an array is accessed. A storage object contains array fields that hold the array elements, and scalar fields that hold data such as the sizes of the array fields. The VOBLA code below defines a storage object for a sparse matrix in the coordinate list (COO) sparse format.

```
storage CooStorage {
  nRows: Index;
  nCols: Index;
  nNonZeros: Index;
  rowIdx: Index[nNonZeros];
  colIdx: Index[nNonZeros];
  data: Value[nNonZeros];
}
```

The COO format stores a matrix as a list of tuples (row index, column index, value). The `nRows` and `nRows` fields store the matrix dimensions and the `nNonZeros` field stores the number of stored tuples. The `rowIdx` and `colIdx` fields contain the row and column indices of the actual matrix elements in the `data` field.

6.3 Layouts

A layout object defines how the fields of a storage are accessed. A layout definition contains the following information:

- The name of a storage object. The fields of the storage can be accessed inside the implementation of layout's methods. Different layouts can use the same storage object.
- A list of interfaces that are implemented by the layout.
- Scalar read-only parameters specific to a particular access pattern. An access pattern cannot be modified and always accesses data in the same way. Parameters can be used to define variations of a given access pattern that iterate over a different number of rows for example.
- An implementation for each of the methods defined in the interface(s).

The following layout defines an access pattern for the COO format:

```
layout Coo: CooStorage implements
            SparseIterable<Value>[][] {
  parameter:
  interface:
    getLen1(): Index { return nRows; }
    getLen2(): Index { return nCols; }

    sparseIterate(): range<Index, Index, &Value> {
      yield rowIdx[k], colIdx[k], data[k]
            forall k in 0:nNonZeros;
    }
}
```

This `Coo` layout refers to the `CooStorage` format defined in Section 6.2 and implements the builtin `SparseIterable` interface. The layout above defines three methods: two `getLen` methods that return the dimensions of the matrix and a `sparseIterate` method that returns the row and column indices and matrix data.

6.3.1 Methods

Layout methods returning scalars or interfaces look similar to regular VOBLA functions. Methods must contain exactly one **`return`** statement at the end of the body in the current implementation of our VOBLA compiler.

Methods of a layout can be called in the method itself via the **`this`** keyword. For example, the `iterate` method below invokes the `access` method defined elsewhere in the layout to obtain the element at position `i`.

```
iterate(): range<Index, &Value> {
  yield i, this.access(i) forall i in 0:n-1;
}
```

Inside layout methods, the **`yield`** keyword is available. Each execution of **`yield`** causes the next tuple of the iteration to be returned.

VOBLA enforces typing rules on methods to ensure they have no side effects. All variables except local variables in a method are read-only. Inside a **`return`** or **`yield`** statement, the arrays defined in the storage become writable such that references can be returned. References to local variables cannot be returned.

6.4 Views

Layouts allow user-defined types but have two shortcomings. First, the user has to define all access patterns for each layout, while some could be reused from other layouts. For example, the `iterate` method of an `Iterable` interface can be implemented from the `[<int>]` and **`len`** methods of an `Accessible` interface. Second, some layouts can be implemented by calling another layout's method in a different way. For example, a layout representing a transposed matrix can be implemented from the layout of the non-transposed matrix by swapping row and column indices.

VOBLA provides views to solve both issues. A view acts as a wrapper around a layout. Instead of being based on a storage, a view is based on an interface called the *base interface*. Methods from the base interface can be accessed using the keyword **`base`**. The view can be used with any layout implementing the base interface to create a new layout. The following VOBLA code defines a view named `TransposedMat` that has base interface `Accessible`. The `access` method is implemented by calling the `access` method from the base interface.

```
view TransposedMat: Accessible<Value>[][]
  implements Accessible<Value>[][] {

  access(i: Index, j: Index): &Value {
    return base.access(j, i);
  }
}
```

This view can then be used in an export statement, to for example obtain a function that accesses a transposed sparse matrix in the COO format:

```
export gemm<Float>(A is TransposedMat(COO), ...
```

6.5 VOBLA Compilation

We have implemented a VOBLA compiler that generates PENCIL code. A key challenge for the VOBLA compiler is that PENCIL does not support pointers or references. To handle references created in interface methods, we inline the interface methods and replace reference variables by the actual memory access they represent. Iterators are compiled in a similar fashion. For example, an `iterate` method defined as follows:

```
iterate(): range<Index, &Value> {
  yield i, data[i] forall i in 0:n-1;
}
```

and used in a VOBLA function as follows:

```
x = i*i forall i, x in X;
```

is translated into the following PENCIL code:

```
#pragma pencil independent
for(int i=0; i<n; i++) {
  data[i] = i*i;
}
```

Since all method calls and iterators are inlined, there is no performance penalty caused by virtual method dispatch overhead. This requires a different version of the function to be generated for each storage format, which allows the PENCIL compiler to separately optimize each function. Although this increases the overall code size, we believe the ability to separately optimize each functions outweighs the code size increase. To enable more optimizations, the VOBLA compiler keeps track of aliasing between function arguments and creates a different version of a function when two arguments are the same.

7. BLAS Library

We have implemented a complete BLAS (Basic Linear Algebra Subprograms) library [9, 12] to demonstrate the practical use of VOBLA. A BLAS library provides basic operations on vectors and matrices such as vector scaling (`scal`), triangular matrix equation solving (`trsv`), and general matrix-matrix multiplication (`gemm`). BLAS functions are commonly used as building blocks for more complex algorithms (*e.g.* LAPACK [13]). BLAS functions are grouped into three levels: level 1 provides vector-vector operations, level 2 provides matrix-vector operations, and level 3 provides matrix-matrix operations. In this section, we show how VOBLA deals with the following two properties of the BLAS interface:

- **Storage layout**: The level 2 and 3 BLAS functions support dense matrix storage layouts and a predefined set of compressed matrix storage layouts such as triangular or symmetric layouts.
- **Substructures**: The BLAS interface allows efficient access to matrix substructures such as rows, diagonals, or submatrices. This is accomplished by providing an increment parameter that selects on which matrix elements a BLAS function iterates.

7.1 Storage Layout

Besides regular dense matrices, the BLAS interface supports a predefined set of compressed matrix storage layouts. For example, the BLAS interface provides symmetric matrix multiplication (`symm`) and hermitian matrix multiplication (`hemm`). These variants differ from `gemm` only in the storage layout of the first input matrix. The matrix multiplication algorithm is identical for each variant. A user first defines these symmetric and hermitian storage layouts in VOBLA, as detailed in Section 6, and then exports the `symm` and `hemm` functions from the `gemm` definition.

To demonstrate the benefits of separating data type, access pattern, and storage layout information from the algorithm, we consider the following generic description of matrix multiplication:

```
function gemm(alpha: Value,
              in A: SparseIterable<Value>[m][k],
              in B: Value[k][n],
              beta: Value,
              out C: Value[m][n]) {
 Cij *= beta forall _, _, Cij in C.sparse;
 C[i][j] += alpha*Ail*B[l][j]
            for i, l, Ail in A.sparse, j in 0:n-1;
}
```

We then use 50 export statements to obtain all BLAS variants for `gemm`, `symm`, and `hemm` from this generic description.

7.2 Substructures

The standard BLAS interface allows efficient access to substructures (such as a row, column, diagonal, or sub-matrix) of a matrix across functions without the need to duplicate matrix elements. The Fortran reference implementation supports such substructures by relying on all matrix elements being stored contiguously in memory as a one-dimensional array. In a such layout, the i-th element of a row, column, diagonal, or sub-matrix is stored at location $\mathit{offset} + i \cdot \mathit{incr}$ of the matrix storage. Here, *offset* is the offset of the first element of the substructure from the beginning of the matrix, and *incr* selects the elements corresponding to the substructure. For example, setting $\mathit{offset} = 2$ and $\mathit{incr} = m$ yields all elements of the second column of an $m \times n$ matrix stored in row-major order. In the reference implementation, *offset* is added to the pointer representing the matrix and *incr* is the increment of a loop iterating over the desired substructure.

Since PENCIL does not support pointers, passing an arbitrary subrange of an array to a function without copying the array data is not trivial. The VOBLA compiler solves this by passing the offset explicitly as an argument, and supporting only row, column, (anti)diagonal, and rectangular submatrix substructures. This solution avoids pointers and array copying, at the expense of not supporting every possible matrix substructure. All information needed to access a substructure is stored in a PENCIL structure that is passed alongside every matrix argument. The PENCIL structure is invisible at the VOBLA level and is generated automatically when a substructure is used in VOBLA.

8. Results

To demonstrate the practical use of VOBLA, we have realized a tool flow to compile our VOBLA BLAS to OpenCL code. We use this tool flow to assess the performance of our BLAS implementation.

8.1 Performance of VOBLA-generated BLAS OpenCL code

We have compiled the VOBLA code for selected BLAS functions into PENCIL code using our VOBLA compiler. We have compiled this PENCIL code into OpenCL code using the PPCG [16] OpenCL backend. A summary of the used BLAS functions and the input data sizes are given in Table 2. All of these BLAS matrix functions operate on dense matrices.

Figure 3 shows the normalized execution time for seven commonly used BLAS functions, on an Arndale board containing an ARM Mali-T604 GPU running at 533 MHz. We compare the VOBLA-default and VOBLA-autotuned implementations to a naive implementation and a hand-optimized implementation. The naive implementations compute one vector or matrix element per work-item and do not incorporate any hand-coded optimizations such as tiling, vectorization, or unrolling. The hand-optimized implementations leverage a range of techniques such as loop tiling, loop fission, and vectorization. Obtaining a hand-optimized implementation requires the programmer to iterate over the source code, gradually improving its execution time while maintaining functional correctness of the code. This is a non-trivial task, and getting good performance requires a thorough understanding of the algorithm, its memory access patterns, and the accelerator platform.

We compare single-precision (32-bit) and double-precision (64-bit) real and complex floating-point implementations for each of the seven BLAS functions. Obtaining the VOBLA-default and VOBLA-autotuned implementations for both data types requires negligible effort, as the programmer only needs to write the desired data type in the **`export`** statement. In contrast, obtaining a hand-optimized implementation for a different data type requires considerable effort. We have found that the optimal implementation for one data

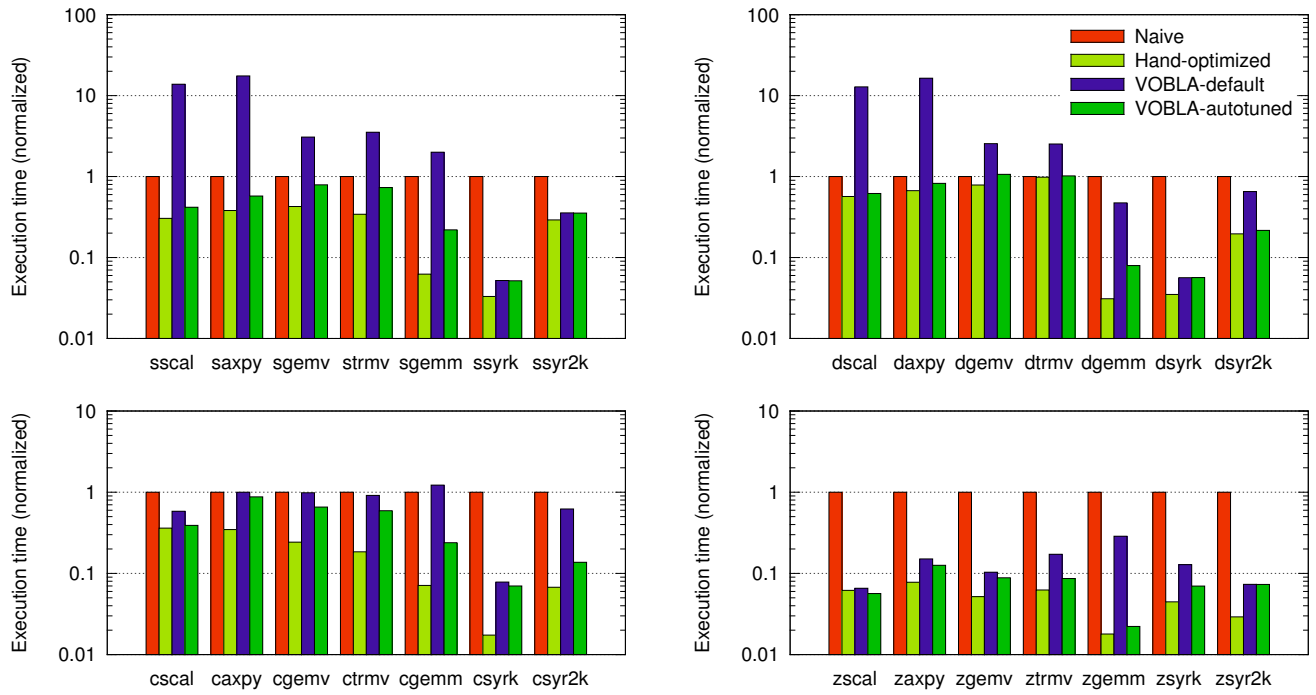

Figure 3. Execution times on an ARM Mali GPU for single-precision (top left), double-precision (top right), complex single-precision (bottom left), and complex double-precision (bottom right) BLAS primitives, normalized to a naive implementation. The VOBLA-default implementations are obtained from PPCG with default parameter values. The VOBLA-autotuned implementations are obtained from PPCG with tuned parameter values found using automated exploration.

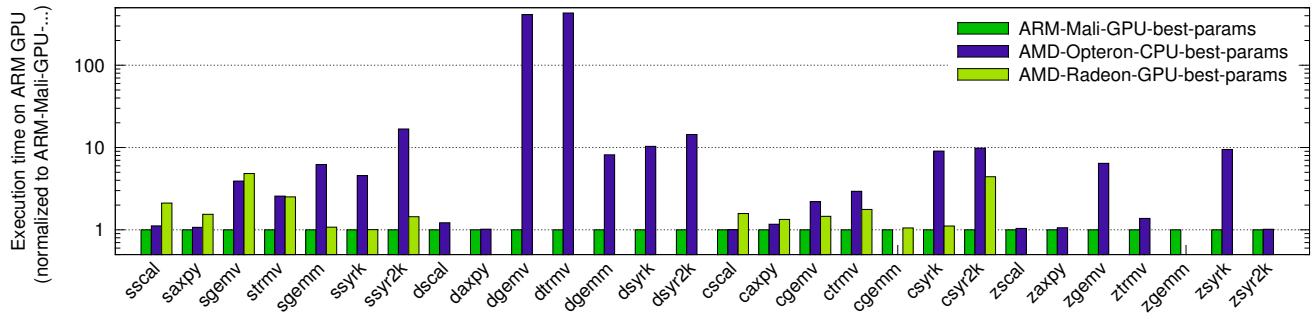

Figure 4. The performance portability problem: the optimal PPCG-generated OpenCL for a particular platform (*e.g.* AMD Radeon GPU or AMD Opteron CPU) is not optimal for another platform (*e.g.* ARM Mali GPU). Results for the double-precision functions (prefixed with *d* and *z*) on the AMD Radeon GPU are missing due to lack of double-precision floating point support on that device.

type is not the optimal implementation for a different data type, as for example the different data size affects cache behaviour considerably. This requires the programmer to iterate over the source code again to obtain an optimized implementation for the new data type.

Currently, the OpenCL backend of PPCG is still under development. In particular, PPCG does not take characteristics of our chosen GPU into account and does not perform vectorization or reduction optimizations. Despite this, we are already able to obtain results competitive with hand-optimized implementations. The hand-optimized implementations currently outperform any of the PPCG-generated implementations on average by a factor of $1.9\times$, with an extremal case of $4\times$ (csyrk). But the programmer effort needed for the hand-optimized implementations is considerably larger than the effort needed for the VOBLA implementations. Moreover, hand-optimized implementations require skills which are not necessarily available to a linear algebra domain expert.

8.2 Performance Portability

Figure 4 illustrates the performance portability problem across three different platforms: an ARM Mali-T604 GPU, an AMD Radeon HD 5670 GPU, and an AMD Opteron 12-core 6164HE CPU. For each BLAS function on each platform, we determine PPCG parameters that result in the lowest execution time. We then generate OpenCL implementations using PPCG with these parameters, and run the OpenCL implementations on a single platform. Figure 4 shows that the best parameters for a particular platform do not give the best performance on another platform. Moreover, the

BLAS Function	Data size	Description
*scal (level 1)	10485760	Vector scale
*axpy (level 1)	10485760	Vector scale and add
*gemv (level 2)	1024×1024	Matrix-vector multiply
*trmv (level 2)	1024×1024	Triangular mat-vec mult.
*gemm (level 3)	1024×1024	Matrix-matrix multiply
*syrk (level 3)	1024×1024	Symmetric rank-1 update
*syr2k (level 3)	1024×1024	Symmetric rank-2 update

Table 2. BLAS Function descriptions.

	Our work	Ref. [12]	uBLAS [17]
Language	VOBLA	Fortran	C++
Passes full test suite	Yes	Yes	Yes[1]
Function count	26	150	24
Total line count	1714	16086	45908
└ **BLAS functionality**	248	16086	158
└ **Data structures**	434	-	45750
└ **Export statements**	1032	-	-

Table 3. Source code statistics for BLAS implementations.

best cgemm and zgemm implementations for the AMD CPU could not be run on the ARM GPU due to tighter resource constraints.

8.3 Productivity

Table 3 shows a comparison of source code statistics for our VOBLA BLAS implementation and two reference implementations [12]. All implementations provide the three levels of the BLAS interface. We break down the line count for the VOBLA implementation into lines describing the core functionality of BLAS functions; lines defining (reusable) data structures such as triangular matrices; and lines containing export statements such that all functions constituting the BLAS interface are generated from the 26 generic VOBLA functions. The VOBLA implementation requires about nine times less lines than the Fortran reference implementation. This is mainly because of the higher level of VOBLA compared to Fortran, and because of the use of generic functions in VOBLA. The line count comparison is even more favourable for VOBLA when omitting the export statements needed to obtain a library conforming to the legacy Fortran BLAS interface.

We also compare the VOBLA implementation to uBLAS, which is a more modern object-oriented BLAS implementation based on C++ templates [17]. Like VOBLA, uBLAS hides the storage format specific operations under the hood, keeping the BLAS functionality code small and clean. However, the data structure definitions require about one hundred times more lines of code than the VOBLA data structure definitions. Although code conciseness is not a guarantee for improved programmer productivity, we believe the significant differences do strengthen our productivity claim.

9. Conclusions

In this paper, we described our work towards solving the *programmer productivity* and *performance portability* concerns for GPUs. We presented VOBLA, a DSL for efficient linear algebra programming. A VOBLA compiler compiles VOBLA programs into PENCIL code. PENCIL has special constructs to carry meta-data that help the polyhedral code generator to generate efficient OpenCL code. We implemented the compilation flow and used it to code the entire BLAS libary in VOBLA. The results showed the VOBLA based BLAS implementation performs $8.1\times$ better than straightforward OpenCL code, although it is still $1.9\times$ slower than hand-optimized OpenCL code for a particular GPU.

[1] Passes internal Boost testsuite for uBLAS, instead of Fortran test suite.

Acknowledgments

This work was partly supported by the EU FP7 STREP project CARP (project number 287767).

References

[1] R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, J. Inoue, and T. Grosser. PENCIL: Towards a Platform-Neutral Compute Intermediate Language for DSLs. *Workshop on Domain Specific Languages, WOLFHPC'12*, 2012.

[2] A. Faucher, C. Fu, D. Callahan, K. Spagnoli, and P. Nagpal. C++ AMP BLAS. `http://ampblas.codeplex.com/`, 2013.

[3] M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison Wesley, 2011.

[4] GNU Project. GSL: GNU Scientific Library. `http://www.gnu.org/software/gsl/`, 1996-2013.

[5] H. Joong, K. J. Brown, A. K. Sujeeth, and H. Chafi. Implementing Domain-Specific Languages for Heterogeneous Parallel Computing. *IEEE Micro*, 31:42–53, October 2011.

[6] K. Goto. GotoBLAS: Texas Advanced Computing Center Software. `http://www.tacc.utexas.edu/tacc-software/gotoblas2`, 2013.

[7] S. Kelly and R. Pohjonen. Worst Practices for Domain-Specific Modelling. *Software, IEEE*, 26(4):22–29, Aug. 2009.

[8] A. Kravets, S. van Haastregt, U. Beaugnon, D. Tweed, J. Absar, and A. Lokhmotov. VOBLA and PENCIL tools. `https://github.com/carpproject`, 2014.

[9] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[10] M. Luján, T. L. Freeman, and J. R. Gurd. OoLALA: an Object Oriented Analysis and Design of Numerical Linear Algebra. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 229–252, 2000.

[11] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, 2013.

[12] The Netlib. BLAS – Basic Linear Algebra Subprograms. `http://www.netlib.org/blas/`, 1979.

[13] The Netlib. LAPACK – Linear Algebra Package. `http://www.netlib.org/lapack/`, 1992.

[14] P. Tillet, K. Rupp, and S. Selberherr. An Automatic OpenCL Compute Kernel Generator for Basic Linear Algebra Operations. In *Proceedings of the Symposium on High Performance Computing*, HPC '12, pages 4:1–4:2. Society for Computer Simulation International, 2012.

[15] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: an Annotated Bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[16] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, Jan. 2013.

[17] J. Walter and M. Koch. uBLAS: Basic Linear Algebra Library. `http://www.boost.org/doc/libs/1_54_0/libs/numeric/ublas/doc/index.htm`, 2013.

[18] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27:2001, 2000.