A Code Generation Framework for Targeting Optimized Library Calls for Multiple Platforms

Tan, Wen Jun; Tang, Wai Teng; Goh, Rick Siow Mong; Turner, Stephen John; Wong, Weng-Fai

2015

Tan, W. J., Tang, W. T., Goh, R. S. M., Turner, S. J., & Wong, W.-F. (2015). A Code Generation Framework for Targeting Optimized Library Calls for Multiple Platforms. IEEE Transactions on Parallel and Distributed Systems, 26(7), 1789-1799.

https://hdl.handle.net/10356/81361

https://doi.org/10.1109/TPDS.2014.2329494

A Code Generation Framework for Targeting Optimized Library Calls for Multiple Platforms

Wen Jun Tan*, Wai Teng Tang‡, Rick Siow Mong Goh‡,
Stephen John Turner*, and Weng-Fai Wong†

Abstract—Directive-based programming approaches such as OpenMP and OpenACC have gained popularity due to their ease of programming. These programming models typically involve adding compiler directives to code sections such as loops in order to parallelize them for execution on multicore CPUs or GPUs. However, one problem with this approach is that existing compilers generate code directly from the annotated sections and do not make use of hardware-specific architectural features. As a result, the generated code is unable to fully exploit the capabilities of the underlying hardware. Alternatively, we propose a code generation framework in which linear algebraic operations in the annotated codes are recognized, extracted and mapped to optimized vendor-provided platform-specific library calls. We demonstrate that such an approach can result in better performance in the generated code compared to those which are generated by existing compilers. This is substantiated by experimental results on multicore CPUs and GPUs.

Index Terms—Code generation, OpenMP, OpenACC, multicore CPU, GPU

1 Introduction

Computing platforms have evolved dramatically over the last decade. Because of the physical limitations imposed by thermal requirements, frequency scaling can no longer be the only mode to increasing the performance of processors. As a result, many hardware manufacturers have turned to scaling the number of cores in a processor in order to boost application performance. Apart from the significantly improved simultaneous multithreading capabilities, these modern processors also contain wide vector processing units to exploit fine-grained parallelism. At the same time, accelerators such as Graphics Processing Units (GPUs) [1], [2] have gained prominence, especially in the high performance computing domain. Many scientific applications from various fields such as economics [3], fluid dynamics [4], molecular dynamics [5] and the social sciences [6] have been ported to use GPUs in order to speed up their computations.

As a result of such hardware trends, the heterogeneity of these platforms and the need to program them efficiently has led to a corresponding explosion in the number of programming models. OpenMP [7], Intel Threading Building Blocks (TBB) [8], Nvidia CUDA [9], OpenCL [10], OpenACC [11], and OpenHMPP [12] are but only a handful out of the many programming models currently available for users to choose from when writing their applications. Usually, the choice of programming model is dictated by the targeted hardware platform. For instance, OpenMP and TBB are typically used for multicore programming, while CUDA and OpenACC are commonly used for GPU programming[1].

Of these models, OpenMP, OpenACC and OpenHMPP are directive-based and are relatively simpler to program compared to TBB or CUDA. In general, directive-based programming models provide an easy way for programmers to parallelize their code. Directives are normally added directly to parallelizable code regions to enable concurrent execution on a target platform. The OpenMP [13] standard is jointly defined and standardized by a group comprising of major computer vendors for targeting shared memory parallel machines, and is supported by compilers from GNU [14], IBM [15], and Intel [16]. Similarly, the OpenACC [17] standard is defined for targeting hardware accelerators such as GPUs, and is supported by compilers from Cray [18], PGI [19], and CAPS [20]. Like OpenACC, OpenHMPP can be used to target GPUs, but is currently only supported by compilers from CAPS and PathScale [21]. In this paper, our focus will be on OpenMP and OpenACC annotated codes[2].

By using OpenMP or OpenACC, existing codes may be ported to multicores or GPUs with less effort by users. These models allow programmers to simply add annotations in the form of compiler directives to regions of codes in order to parallelize them for execution on either multicores or GPUs. In many

- *School of Computer Engineering, Nanyang Technological University, Singapore
- †Department of Computer Science, School of Computing, National University of Singapore
- ‡Institute of High Performance Computing, Agency for Science, Technology and Research, Singapore

1. Note that OpenMP is not exclusively for multicores and OpenACC is not GPU-specifc only.

2. As OpenACC is supported by more compiler vendors, we picked OpenACC and left out OpenHMPP in our study.

cases, there is no need to modify the data structures or make architectural changes to the underlying code base when adding the directives. Although easier to program, there is a problem with programming using these directive-based models. Many directive-annotated loops contain abstractions that have equivalents in vendor-provided libraries which are highly optimized for each particular platform. However, many existing compilers do not generate codes which utilize these highly optimized libraries. To solve this problem, we demonstrate a code transformation and generation framework that is able to translate user codes to high level mathematical operations and then map them directly to efficient and optimized routines on the target platforms. The operations that are under the scope of this work are linear algebraic operations. The purpose of our proposed framework is not to replace the directive-based approach of OpenMP or OpenACC, but to complement it for optimized code generation.

The rest of the paper is organized as follows. We first discuss prior work in Section 2. Then in Section 3, we discuss the motivation behind our work. Section 4 and Section 5 present our proposed code generation framework and its components in greater detail. In Section 6, we present experimental results and demonstrate the relevance of our proposed framework. Finally, we conclude our paper in Section 7.

2 Related Works

Previous code generators or compilers such as PetaBricks [22], SPIRAL [23], and Delite [24] can be used to target multicores or GPUs. PetaBricks [22] is an implicitly parallel language and compiler where the user is able specify multiple implementations of an algorithm. The PetaBricks compiler then autotunes programs by making fine-grained algorithmic choices given the user-specified algorithmic implementations. The SPIRAL program generation system [23] is a domain-specific library generator that automates the production of high performance code for digital signal processing (DSP) transforms. It does this by exploiting the mathematical structure of DSP transform algorithms to search for the best algorithmic and implementation choice for different computer architectures. The Delite Compiler Framework and Runtime [24] targets heterogeneous execution by providing a framework for expressing algorithms through the use of an embedded domain-specific language.

Many of these systems require the user to implement the algorithms using a proprietary language or API. On the other hand, our proposed framework only requires the user to provide directive annotated code to be able to target multicores or GPUs. Furthermore, in order to benefit from platform-specific optimizations, our focus is on identifying and extracting well-defined linear algebraic forms through a matching algorithm and mapping them to optimized libraries instead of direct code generation.

Also related to our work is the BLAS-like Library Instantiation Software (BLIS) framework [25]. BLIS is a new infrastructure that aims to address the shortcomings with the current BLAS interface by expressing the BLAS operations in terms of simpler kernels. However, BLIS currently does not implement multithreading, nor support GPUs. Another different work seeks to update BLAS by extending it with additional functionalities [26]. Build-to-order BLAS [27] and Design-by-transformation BLAS [28] approach the problem from a different angle. Their goal is to generate optimized and tuned BLAS-like functions from high level kernel specifications.

Our code recognition and generation framework differs from these works in a few ways. First, BLAS represents only a subset of the possible linear algebraic operations which our framework recognizes. Hence, once BLAS extensions are available, it is relatively straightforward to extend our framework to provide support for them. Second, these approaches attempt to construct better libraries for an application. Our work, on the other hand, aims to transform the application to better fit the available libraries. In other words, our approach complements these works in that our framework can be made to generate code to make use of any optimized BLAS library that is provided. For example, our framework can be extended to make use of other BLAS packages such as PLASMA or MAGMA [29]. The former targets multicore systems whereas the latter targets heterogeneous architectures.

3 Code Transformation Framework

The directive-based approach taken by both OpenMP and OpenACC provides a simple but powerful way for users to quickly parallelize their code (for convenience, we shall use the C language form of the compiler directives for all the examples in the rest of the paper). Users typically need to identify portions of their code which are amenable for parallel execution and annotate them as *parallel* sections. For instance, `#pragma omp parallel` is used to create parallel code sections for multicore execution. In the case of OpenACC, the annotation `#pragma acc parallel` or `#pragma acc kernels` can be used to generate GPU specific code for a given code section. In many instances, code sections containing nested loops are usually good candidates for parallelization on multicores or GPUs, and are annotated using `#pragma omp parallel for` and `#pragma acc kernels loop` for OpenMP and OpenACC, respectively.

Nevertheless, the current directive-based approach has a few shortcomings in terms of the code generated. Consider the code fragment written in C in Figure 1a. This code fragment computes a symmetric matrix `B` by adding `A` to its transposed version. A

```
1 for (int i = 0; i < N; i++)
2 {
3   for (int j = 0; j < N; j++)
4   {
5     B[i*N+j] = A[j*N+i] + A[i*N+j];
6   }
7 }
```

(a) Original code fragment

```
1 #pragma acc kernels loop \
2   independent collapse(2)
3 for (int i = 0; i < N; i++)
4 {
5   for (int j = 0; j < N; j++)
6   {
7     B[i*N+j] = A[j*N+i] + A[i*N+j];
8   }
9 }
```

(b) After adding OpenACC pragmas

```
1 float alpha = 1;
2 float beta = 1;
3 cublasSgeam(handle, CUBLAS_OP_T,
4   CUBLAS_OP_N, N, N, &alpha, devA,
5   N, &beta, devA, N, devB, N);
```

(c) Using optimized library call

Fig. 1. An example illustrating the advantage of our proposed code transformation framework.

typical way to execute this on the GPU is to annotate it with OpenACC pragmas as shown in Figure 1b. An OpenACC compiler such as the PGI Accelerator compiler [19] analyzes the annotated code and transparently maps the loops to grids, thread blocks and warps according to an internal cost model[3]. However, many compilers currently do not generate very optimized binaries for such codes. This is because architectural-specific features such as shared memory is usually not exploited by the compiler in the generation of the compiled code. In contrast, platform-specific libraries usually contain highly optimized routines that can be used to replace a matching code fragment.

For instance in the example in Figure 1b, one can replace the code fragment with a corresponding CUBLAS `geam` function call that is well optimized and takes advantage of per-processor shared memory. An equivalent code that makes use of the optimized library call is shown in Figure 1c. To demonstrate the difference in performance between directive-generated code and platform-specific library call, Figure 1b is compiled with PGI Compiler version 13.10 while the code in Figure 1c is compiled using the compiler from Nvidia SDK 5.0. Both codes are executed on a Nvidia Tesla K20 GPU. Table 1 shows our experimental configuration. For a value of $N = 1024$, empirical results indicate that the code in Figure 1c runs 2.5 times faster than the corresponding code in Figure 1b.

3. Nvidia uses the terms *grid*, *thread block* and *warp* which are synonymous to the terms *gang*, *worker* and *vector* used in OpenCL terminology.

TABLE 1
Hardware configuration.

	CPU	GPU
Architecture	Intel Xeon 2 x E5-2665	Nvidia Tesla K20
Number of cores	16	2496
Clock rate	2.40GHz	706MHz
Size of memory	32GB	5GB
Memory bandwidth	8GB/s	208GB/s

Therefore in this paper, we will describe and evaluate a source-to-source code transformation framework that is able to extract such known abstractions and directly map them to efficient vendor-provided routines. In this work, we restrict ourselves to the class of linear algebraic operations, with which we will demonstrate our approach since most hardware platforms have optimized libraries provided by vendors for this class of operations.

Figure 2 shows how our code transformation framework works. Central to the framework is the *Extraction Engine*. The Extraction Engine analyzes loop nests that can be annotated by OpenACC or OpenMP, and performs recognition of known abstractions according to a matching algorithm which will be described later. The abstractions are then extracted and passed to the *Mapping Engine*. The Mapping Engine contains an optimization stage which attempts to fuse the extracted operations together in order to minimize the number of library calls. Finally, the engine transforms them into platform-specific library calls and generates the appropriate code according to the target platform.

The approach taken by this framework has several advantages. First, it allows users to continue to express their code with loop nests and annotate them with pragmas. There is no need to learn a new programming model. Second, it enforces separation of concerns between writing idiomatic code and writing low-level optimized code; the user need not be concerned with optimization as it is the responsibility of the framework to recognize the code and generate optimized library calls. Last but not least, because optimization and mapping is delegated to the framework, the framework can be updated without changes to the original code. For example, CUDA versions prior to 5.0 did not have the `geam` function, therefore the framework generated kernels directly from the code, similar to what other compilers do. Once `geam` was included in CUDA 5.0, the framework

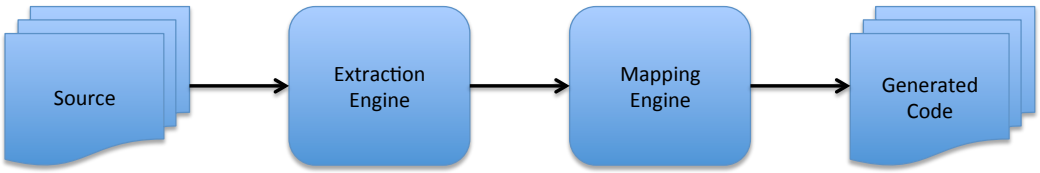

Fig. 2. A flow diagram of proposed code generation process. The *Extraction Engine* performs matching and recognizes linear algebraic operations from the code, whereas the *Mapping Engine* optimizes and maps them to the equivalent library calls for a target hardware platform.

was updated and is able to generate direct calls to `geam`, without requiring the user to modify any of the original source codes.

4 The Extraction Engine

In this section, we describe the Extraction Engine in greater detail. The purpose of the Extraction Engine is to take a loop nest and identify all the parts that can be translated into optimized library calls. It does this by a series of steps that include extracting critical information from the code and performing pattern matching on the expressions to determine the linear algebraic operations. Once the linear algebraic operations have been extracted from the loop nests, they are forwarded to the Mapping Engine to be optimized and mapped into the appropriate library calls. The extraction and recognition process consists of the following steps:

1) Compute the domain for each array in a statement and identify the shape of the operands (see Sections 4.1.1 and 4.1.2).
2) Perform preprocessing to reduce program variations to allow for better pattern matching (see Section 4.1.3).
3) Perform recognition of linear algebraic operations (see Section 4.2).

The first two steps are preprocessing stages while the third step is performs the actual algorithmic recognition. These steps are described in greater detail in the following sections. For step 1, Section 4.1.1 describes the domain computation, and Section 4.1.2 describes shape recognition of the operands. For step 2, Section 4.1.3 describes how to perform canonical reduction before recognition is carried out. Finally for step 3, Section 4.2 describes the procedures for recognizing linear algebraic operations.

4.1 Preprocessing

4.1.1 Domain Information Extraction

In the initial step, it is crucial to first identify domain related information for each statement in a loop nest. The Extraction Engine internally employs a polyhedral representation of the given loop nest [30], [31]. Critical information for each statement consists of the set of array references and their associated iteration domain (D^S), schedule (Θ^S) and access function ($f(\cdot)$). A loop nest with depth n is represented with a n-entry column vector called the *iteration vector*; the loop nest also defines the *iteration domain* of a given statement. Together with the set of local variables, global parameters, and iteration vectors, one can setup a set of linear equalities which define the polyhedron for a given statement. Let a statement S be surrounded by loops with a depth n, then $\mathbf{i}$ is an iteration vector which contains the loop indices and has a dimension of n. Further, let $\mathbf{i}_{lv}$ and $\mathbf{i}_{gp}$ be the vector of local variables and global parameters on which the loops depend, and let Λ^S be the matrix of linear constraints which bound the domain of the iteration, local variable and global parameter vectors, then the iteration domain of S is defined by

$$D^S = \{\mathbf{i} | \exists \mathbf{i}_{lv}, \Lambda^S \times [\mathbf{i}, \mathbf{i}_{lv}, \mathbf{i}_{gp}, 1]^t \geq 0\}. \quad (1)$$

Also associated with the statement S is its schedule, which essentially defines *timestamps* for each statement instance using the iteration vector and statement order in order to accommodate loop nests with multiple statements. For example, the schedule matrix Θ^S is defined such that an instance of S given by a particular $\mathbf{i}_k$ is executed before another instance $\mathbf{i}_{k'}$ of statement S' if and only if

$$\Theta^S \times [\mathbf{i}_k, \mathbf{i}_{lv}, \mathbf{i}_{gp}, 1]^t \ll \Theta^{S'} \times [\mathbf{i}_{k'}, \mathbf{i}_{lv}, \mathbf{i}_{gp}, 1]^t \quad (2)$$

For each statement S, we define a set of (A, f) pairs where each pair represents a reference to the variable A in the left or right hand side of the statement and f is the *access function* which maps iterations in D^S to elements accessed by the variable A. Note that f is a function of the loop iteration vector, local variables and global parameters. The access function f can be defined by a matrix F such that

$$f(\mathbf{i}) = F \times [\mathbf{i}, \mathbf{i}_{lv}, \mathbf{i}_{gp}, 1]^t. \quad (3)$$

4.1.2 Shape Recognition

As the iteration space of loop nests are sets of integer-valued points in regions of spaces, the distribution of the points can be used to deduce the shape of array expressions in each statement. For each (A, f) pair,

the domain of each array expression of variable A is defined as D^A, which is obtained by subjecting the access function f to the iteration domain D^S imposed by Equation 1. If D^A only consists of linear constraints, where each constraint is an affine expression containing only a single loop index, we can deduce that the array expression is a rectangular region. If the linear constraints for each loop only contains a lower bound and upper bound, the number of rows in D^A gives the dimensionality of the array expression, which identifies the array expression as a vector or matrix. Given that the shape of the array is defined, we can identify whether the array expression is a subarray or a slice of the original array A.

For example, the iteration domain of a statement S is shown in Equation 4. An array A with the expression `A[i-1]` has the access function $\begin{bmatrix}1 & 0 & -1\end{bmatrix}$. By applying the access function to Λ^S gives the accessed domain D^A in Equation 5. Since the linear constraints only contains i, A is identified as a vector with the domain: $0 \leq i \leq N-2$.

$$\Lambda^S = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \end{bmatrix} \begin{array}{l} 0 \leq i \\ i \leq N-1 \end{array} \tag{4}$$

$$D^A = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -2 \end{bmatrix} \begin{array}{l} 0 \leq i \\ i \leq N-2 \end{array} \tag{5}$$

Currently only general square matrices are recognized, the recognition can be extended to other matrices such as symmetric, triangular and Hermitian matrices, as well as banded and packed matrix storage format.

4.1.3 Canonical Reduction

Before performing algorithmic recognition in the next step, we perform reduction of the loop nest into a canonical form. The first step is to perform loop fission on the collection of statements in the loop nest. This transformation is applied at all loop depths in the nest and serves to split all statements to the maximum extent allowed by the dependencies across the statements. That is, if there are no flow dependencies and loop-carried dependencies between two statements, then it is possible to break them up while preserving their nesting depth. Otherwise, they are retained in the same block. The loop fission transformation only modifies the statement schedule but does not change their dependencies in any way. Details about the transformation can be found in [31]. Validity of the transformation is verified by performing dependence checking using the polyhedral model [32]. In the next step of reduction, the expression tree for each statement is normalized into canonical form through a lexicographic sort, by sorting the operators within the expression according to the operator precedence table and then grouping terms with the same operators together. This reduces variations in which expressions can be written. Furthermore, each term in the expression tree is initialized with an associated domain information that is related to its enclosing statement and determined previously by $\{D^S, \{(A, f)\}\}$. This will allow us to perform recognition using the rules that are described next, and to propagate the domain information up the abstract syntax tree.

Iterators $\{i, j\}$ on left hand side and $\{i\}$ on right hand side.

```
1  for(i=0; i<n; i++)
2  {
3    for(j=0; j<n; j++)
4    {
5      A[i][j]=x[i];
6    }
7  }
```

Linear algebraic form after extending with outer product:
$A \leftarrow x \otimes 1_n$

Fig. 3. Code fragment that requires the outer product operator to extend the domains.

Iterators $\{i, j\}$ on left hand side and $\{j, i\}$ on right hand side.

```
1  for(i=0; i<n; i++)
2  {
3    for(j=0; j<n; j++)
4    {
5      A[i][j]=B[j][i];
6    }
7  }
```

Linear algebraic form after applying permutation to the indices:
$A_{i,j} \not\leftarrow B_{j,i} \Rightarrow A_{i,j} \leftarrow (B^T)_{i,j}$

Fig. 4. Code fragment that permutes the indices to match the domains.

4.2 Algorithmic Recognition

In this step, the high-level abstractions are recognized and converted into linear algebraic form. Lifting the statement to a high-level abstraction relies on matching the expression tree of a single statement with a series of rules. Matching is performed on the abstract syntax tree by traversing from the leaf nodes at the bottom to the top of the tree. There are two sets of matching rules based on the type of assignment operators. Each statement will be abstracted using the **Assignment** rule if the assignment operator is non-reducing (e.g. =), otherwise the **Reduction** rule is used if the assignment operator is of a reduction type (e.g. +=). Note that for each statement, the domain

TABLE 2
Dimension of D^S_{lhs} determines the operator under the Reduction rule.

Dimension of D^S_{lhs}	Matched Operator
0	Dot Product ($\bullet$)
1	Matrix-Vector Multiply ($\times$)
2	Matrix-Matrix Multiply ($\times$)

D^S has already been determined previously from the polyhedral representation.

According to **Assignment** rule, we can determine the domains on the left hand side of the assignment, D^S_{lhs}, and domains on the right hand side of the assignment, D^S_{rhs}. Subsequently, the right hand expression tree is traversed and replaced with an equivalent expression such that the domains on the right hand side, D^S_{rhs}, is equal to D^S_{lhs}. However, if the domains do not match, there are two possible transformations to assist in the matching:

1) If the dimension of the variables on each side of the assignment does not match, the outer product operator is used to extend the domain for the right hand expression.
2) If the order of domains of each variable does not match, the vector of loop indices (i) is permuted to obtain the domains in the same order.

An example that demonstrates the need for adding the outer product operator is shown in Figure 3. Figure 4 shows permutation of the loop indices with a permutation operation to obtain domains with the same order.

For the **Reduction** rule, we also define the *reduction* domain, R^S, which contains domains that are to be reduced. This can be determined by the set difference in the domains between each side of the assignment, D^S_{lhs} and D^S_{rhs}. A reduction operator such as additive reduction is returned after appropriate permutation of R^S. Subsequently for the addition reduction operation, additional matching based on rules for inner product is performed on the right hand expression. If the top level operator on the right hand side expression is multiplication, the expression returned is determined by the dimension of D^S_{lhs}, as shown in Table 2.

The benefit of using such a technique is that there is no need to perform matching on the whole abstract syntax tree. The extent of this algorithm applies to single statements in a loop. Statements with dependencies across loop iteration are not matched. In addition, variables in a statement do not need to be renamed during matching. Only the domains and operators in a statement are used in the pattern matching. Therefore, there is flexibility in matching the domain, allowing permutation in the ordering of the indices. This method is not restricted to any programming model as long as the language supports arrays and loops. Other languages such as Matlab can be supported in the future. Once the high-level linear algebraic forms have been recognized by the Extraction Engines, these are then fed to the Mapping Engine for code generation as described in the next section.

TABLE 3
BLAS Operations

Sum	$r \leftarrow \sum_i x_i$	SUM
Scaled vector addition	$y \leftarrow \alpha x + y$	AXPY
Copy	$y \leftarrow x$	COPY
Dot product	$r \leftarrow x \bullet y$	DOT
Scaling	$x \leftarrow \alpha x$	SCAL
Matrix-vector product	$y \leftarrow \alpha A \times x + \beta y$	GEMV
Outer Product	$A \leftarrow \alpha x \otimes y + \beta A$	GER
Matrix-matrix product	$C \leftarrow \alpha A \times B + \beta C$	GEMM

TABLE 4
BLAS-like Extensions

Vector zeroing	$x \leftarrow 0_n$	XZERO
Scaled vector accumulation	$y \leftarrow \alpha x + \beta y$	AXPBY
Matrix zeroing	$A \leftarrow 0_{m,n}$	MATZERO
Matrix copy	$B \leftarrow A$	MATCOPY
Scaled matrix	$B \leftarrow \alpha A + \beta B$	MATSCAL
Scaled matrix accumulation	$C \leftarrow \alpha A + \beta B$	MATACCSCAL

5 Mapping Engine

After the extraction phase has identified the high level operations, the Mapping Engine is then invoked to optimize them and translate them to equivalent library calls that are supported by vendor libraries. Note that because the class of linear algebraic operations is a superset over BLAS operations, many vendor libraries have provided additional library functions which are called BLAS-like extensions.

BLAS operations are classified into three different levels – Level 1 performs vector operations, Level 2 performs matrix-vector operations and Level 3 is for matrix-matrix operations. Table 3 shows a list of the BLAS operations that are mapped by our framework. In addition, additional mappings are supplemented in the form of BLAS-like extensions and are listed in Table 4. A, B and C denote two-dimensional matrices while x, y and z denote one dimensional vectors. The Mapping Engine performs the actual code transformation using the following steps: (a) statement fusion (b) operation mapping, and (c) platform-specific code generation.

5.1 Statement Fusion

Because many of the linear forms listed in Tables 3 and 4 are made up of compound operations, for

Original statements:

$$x_{old} \leftarrow x \quad (6a)$$
$$x \leftarrow 0_n \quad (6b)$$
$$x \leftarrow T * x_{old} + x \quad (6c)$$
$$x \leftarrow c + x \quad (6d)$$

After constant propagation:

$$x_{old} \leftarrow x \quad (7a)$$
$$x \leftarrow T * x_{old} \quad (7b)$$
$$x \leftarrow c + x \quad (7c)$$

Final fused form:

$$x_{old} \leftarrow x \quad (8a)$$
$$x \leftarrow T * x_{old} + c \quad (8b)$$

Fig. 5. An illustration of statement fusion.

instance the outer product form contains scaling, addition and the outer product operator, it is necessary to optimize the extracted high level operations by fusing them together into the appropriate linear form in order to reduce the number of library calls. This step is done before the actual mapping is performed. By performing fusion of the statements, it is thus possible to chain them up into more complex operations which can then be mapped and completed in one single library call. Three levels of fusion have been implemented: *fuse constant (FUSE-C)*, *fuse intermediate (FUSE-I)* and *fuse terms (FUSE-T)*. *FUSE-C* essentially performs constant folding and constant propagation to the statements using an extension of standard flow analysis methods [33]. The difference is that instead of applying to scalar variables only, it is extended to take vectors and matrices into account during the analysis. *FUSE-I* performs fusion and eliminates terms that are used as intermediate arrays in the program. *FUSE-T* performs substitution of a term by its corresponding expression.

Figure 5 shows an example code fragment and the result after the fusion process has been applied on it. After constant propagation, the original statement $x \leftarrow 0_n$ in line 6b (0_n denotes the zero vector) is fused with the x on the right-hand-side of line 6c, resulting in $x \leftarrow T * x_{old} + 0_n$, which is then further simplified to $x \leftarrow T * x_{old}$. Further fusion is performed on lines 7b and 7c to remove the intermediate x term which is only used once in the code fragment, resulting in the final form in line 8b.

5.2 Operation Mapping

After fusion, the next step performs mapping of the linear algebraic forms to equivalent BLAS or BLAS-like routines. This is done by stepping through a series of predefined rewrite rules to generate the final

TABLE 5
List of patterns and corresponding operation rules.

Match Operator/Pattern	Rule
$\bullet$	**Dot Product**
$\times$	**Matrix Multiply**
$\otimes$	**Outer Product**
$\alpha a + \beta b$	**Scaled Addition**

mapping. There are two levels of rewrite rules: (a) statement rule and (b) operation rules. A statement rule contains a combination of operation rules. It combines a number of operation rules according to the operator or expression pattern on the right hand side of a statement. Table 5 shows a list of patterns to be matched and the resultant operation rules to be executed, ordered according to the precedence shown in the table.

Since a statement may contain a number of linear algebraic operations, it is necessary for the statement rule to identify and break up a statement into a number of constituent operations based on the operators in the previous table. This is achieved by a bottom up inspection of the operators in the abstract syntax tree, and matching them with the operators listed in the Table 5. As an example, consider the statement $C \leftarrow A \times B + x \otimes y$. Following the precedence defined in the table, this statement is split into two constituent operator rules using **Matrix Multiply** and **Outer Product** to give $C \leftarrow A \times B$ and $C \leftarrow x \otimes y + C$.

Once a statement rule has been determined and the operation rules identified, the corresponding operation rules are executed subsequently. The goal of the operation rules is to match the library call and to identify its calling parameters. The following operation rules are defined:

Dot Product Rule. This rule returns the `DOT` operation. The rule only accepts two operands, i.e. it disallows chaining of dot products.

Outer Product Rule. This rule returns the `GER` operation and supports only two operands. It also performs matching for α and β. In particular for β, three forms of the `GER` operation are returned as a result of this rule as shown in Table 6.

Matrix Multiply Rule. This rule segments the expression into a binary expression with two operands, A and B, and results in either a matrix-vector or matrix-

TABLE 6
Different forms of `GER` depending on the value of β.

Pattern	Result
$A \leftarrow \alpha x \otimes y + \beta A$	`GER`
$A \leftarrow \alpha x \otimes y + A$	`GER`, $\beta \leftarrow 1$
$A \leftarrow \alpha x \otimes y$	`GER`, $\beta \leftarrow 0$

TABLE 7
Different forms of GEMV and GEMM depending on the dimensions of A and B.

Condition	Result
A is 1D and B is 1D	GEMV
A is 2D and B is 1D	GEMV
A is 2D and B is 2D	GEMM

TABLE 8
Different scaled vector operations depending on the value of β.

Pattern	Result
$y \leftarrow \alpha x$	SCAL
$y \leftarrow \alpha x + y$	AXPY
$y \leftarrow \alpha x + \beta y$	AXPBY

matrix operation depending on their dimensionality determined by the Extraction Engine. The equivalent BLAS operations GEMV and GEMM accept only two operands. This rule is shown in the Table 7.

Sum Rule. This rule performs matching of the sum reduction operation and returns the SUM operator.

Scaled Vector Rule. This rule generates operations based on variations of the vector scaling operation. The rule returns different library calls according to the value of β, as shown in Table 8.

Scaled Addition Rule. This rule performs matching of the following statement, $c \leftarrow \alpha a + \beta b$, which is a scaled addition of a and b. This rule is more complex as it allows matching several different operations which contain scaling of terms, such as $C \leftarrow \alpha A \times B + \beta C$ or $y \leftarrow \alpha x + y$. For example, if $a \rightarrow A \times B$, the rule executes the **Matrix Multiply** rule to determine the exact operation to be generated according to the dimension of A and B. On the other hand, if $a \rightarrow x \otimes y$, the rule **Outer Product** is executed. Furthermore, if c is a vector, the **Scaled Vector** rule is called to identify different versions of the scaled vector operation. Otherwise, if c is a matrix, the rule checks if b is scalar variable and if so, the final operation will be determined as a constant addition to a matrix (MATADDCONST). Otherwise, the expression is mapped to an matrix accumulation operation (MATACCSCAL). A summary of the patterns is shown in Table 9.

TABLE 9
Pattern matching with the Scaled Addition Rule.

Pattern/Condition	Result
$a \rightarrow A \times B$	match rule **Matrix Multiply**
$a \rightarrow x \otimes y$	match rule **Outer Product**
c is 1D	match rule **Scaled Vector**
a is 2D	returns MATACCSCAL
b is scalar	returns MATADDCONST

TABLE 10
Mapping of BLAS-like extensions to MKL and CUBLAS routines

BLAS	MKL	CUBLAS
AXPBY	axpby	geam
MATZERO	imatcopy	geam
MATCOPY	omatcopy	geam
MATSCAL	imatcopy	geam
MATACCSCAL	omatadd	geam

5.3 Platform-Specific Code Generation

In the final step, platform-specific code generation is performed based on the resultant operations identified in the previous stage. These operations are mapped to vendor-specific libraries that contain BLAS or BLAS-like library calls. Because BLAS-like extensions differ among different vendor-provided libraries, if a particular vendor library does not provide a specific routine, we substitute the library call with our internal implementation. These are simple operations that involve a vector or matrix with a scalar value, including addition, assignment, exponential and scaling (see Section 5.3.2 for an example). In the following, we provide details about the mapping of BLAS-like extensions to two specific vendor-provided libraries, Intel MKL [34] and Nvidia CUBLAS [35], as they are widely used.

5.3.1 *Intel MKL*

Apart from the standard BLAS functions, Intel MKL provides several routines to extend the functionality of its library. This includes routines to perform data manipulation, such as in-place and out-of-place matrix transposition combined with simple matrix arithmetic operations. Each routine adds the possibility of scaling during the transposition operation through the α and β parameters. For example, imatcopy performs scaling and in-place transposition of matrices; omatcopy performs scaling and out-of-place transposition of matrices, and omatadd performs scaling and addition of two matrices including their out-of-place transposition. The mapping to the Intel MKL routines are shown in Table 10. To perform a MATZERO operation, the scaling parameter α for A can be assigned to zero. For copying, when the scaling factor α is assigned to one, *omatcopy* essentially performs a matrix copy.

5.3.2 *Nvidia CUBLAS*

CUBLAS provides BLAS-like extensions through the following operations shown in Table 10. The geam

```
for(i = 0; i < N; i++)
{
  C[i] = pow(A[i], b);
}
```

(a) Original code

```
__device__ void VecAdd1D(float* A,
  const float b, float* C)
{
  int i = threadIdx.x;
  C[i] = pow(A[i], b);
}

// Kernel call
VecAdd1D<<1,N>>(A, b, C);
```

(b) Generation of a one-dimensional sub-kernel

Fig. 6. Generation of non-standard function into CUDA sub-kernels.

function performs matrix-matrix addition with optional transposition of the matrices. A matrix can also be reset to zero by setting α and β to zero, whereas matrix copying can be achieved by setting $\alpha \leftarrow 1$ and $\beta \leftarrow 0$. In addition, certain statements may contain functions which do not have an equivalent library call. For such statements, we generate CUDA sub-kernels directly. For example in Figure 6a, there is no library call that supports the power function `pow`. Therefore, a sub-kernel as shown in Figure 6b was generated. Essentially, sub-kernels are defined as CUDA device-only functions and they are generated at the granularity of a basic block.

We also convert OpenACC data copy directives to the corresponding CUDA memory copy calls for transferring data between the host and the GPU. These directives are user annotated. The relevant directive is `#pragma acc data`, and the clauses supported are `copy`, `copyin`, `copyout`, and `create`. `copy` is used for two way transfers between the host and the GPU. `copyin` and `copyout` are for one way transfers into and out of the GPU respectively. `create` is used for allocating memory on the device only. The first three directives are translated into their associated `cudaMemcpy` calls, whereas the last directive is translated to a `malloc` call on the device.

6 Evaluation and Results

Using our proposed framework, C or C++ code annotated with directives can be mapped to optimized vendor-provided library calls for different hardware targets. In our experiments, two possible platforms are supported by the framework: Intel MKL for multicore CPUs and Nvidia CUBLAS for GPUs. Codes are generated to the corresponding interfaces provided by these two libraries. We will refer to codes generated by the framework as *Framework Generated Code* (FGC). FGC-MKL and FGC-CUBLAS will be used to denote codes generated for Intel MKL [34] and Nvidia CUBLAS [35] by the framework respectively.

TABLE 11
Summary of parallelizable loops in benchmarks that were annotated and number of library calls generated.

Benchmark	Size	Loops	Annotations	Library calls
cg	5000	14	13	13
gmres	5000	28	18	12
jacobi	20000	3	3	4
matexp	2000	18	10	6
mcl	1000	14	10	8
qr	1000	17	11	11

6.1 Setup

We conducted experiments on a workstation with two 8-core Intel Xeon E5-2665 and a Nvidia Tesla K20 GPU. Details regarding the configurations can be found in Table 1. The number of threads assigned for the MKL experiments was 16 and corresponds to the number of physical cores available. The following codes were used in our evaluation, and were taken from online sources[4]:

- *cg* — Conjugate Gradient method, without preconditioning, for solving a symmetric and positive-definite system of linear equations.
- *gmres* — Generalized Minimal Residual method, without preconditioning, which uses an iterative method to solve a non-symmetric system of linear equations.
- *jacobi* — A classical stationary iterative algorithm for solving a system of linear equations.
- *matexp* — Computation of the matrix exponential.
- *mcl* — Markov Cluster algorithm, a fast and scalable unsupervised cluster algorithm for graphs.
- *qr* — A matrix decomposition algorithm that is used in solving linear least squares problems.

For these benchmark codes, we added compiler directives to the parallelizable loops, targeting multicore CPUs with OpenMP and GPUs with OpenACC. Table 11 shows for each of the benchmarks, the size of input matrices, the number of loops

4. *cg* and *gmres* were taken from the Iterative Methods Library (IML++, http://math.nist.gov/iml++/), *matexp* is from the GNU Scientific Library (GSL, http://www.gnu.org/software/gsl/), *jacobi* is taken from http://people.sc.fsu.edu/~jburkardt/vt2/fsu_open_mp_2008/jacobi/jacobi.html, *qr* is taken from http://mazack.org/code/index.php, and *mcl* is part of the MCL-edge network analysis tools (http://www.micans.org/mcl/).

TABLE 12
Speedup (>1) or slowdown (<1) relative to un-annotated C code (column *Original*) compiled with ICC (*-opt-matmul* flag). For CPU performance, comparisons are made between OpenMP directive and FGC-MKL compiled codes. On the GPU, comparisons are made between OpenACC directive and FGC-CUBLAS compiled codes. Run times are recorded in milliseconds.

Benchmark	Original	OpenMP		FGC-MKL		OpenACC		FGC-CUBLAS	
	Time	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
cg	31,659	24,191	1.31	21,964	1.44	8,019	3.95	8,021	3.95
gmres	30,566	33,325	0.92	27,292	1.12	19,435	1.57	15,974	1.91
jacobi	2,546	1,528	1.67	1,463	1.74	673	3.78	669	3.80
matexp	876	1,207	0.73	336	2.61	1,073	0.82	59	14.63
mcl	12,055	17,732	0.68	1,812	6.65	18,883	0.64	10,628	1.13
qr	3,423	792	4.32	123	27.75	243	14.07	90	38.03

in the code, the number of loops that can be parallelized using compiler directives, and the number of library calls that were generated by the framework.

To evaluate our framework, experiments were conducted to compare performances on both the CPU and GPU. On each hardware target, we compare the performance between directive generated codes and our framework generated codes (FGC). Each benchmark is executed for 10 runs, and the median of the execution time for each kernel is recorded. For the CPU benchmarks, only the execution time of the kernel is recorded; a warm-up code is added to preload any required libraries to reduce the initialization overhead. For the GPU benchmarks, the recorded time includes the data transfers between the host and the GPU. On the CPU, our framework generates MKL library calls (FGC-MKL), and the codes are compiled with Intel Compiler (version 14.0.1) and linked with the multi-threaded Intel MKL (version 11.1). On the GPU, the framework generates CUBLAS calls and the codes are linked to the Nvidia CUBLAS 5.0 library.

6.2 Performance Comparison

In this section, we compare the performance of the compiler generated codes to the codes generated using our recognition and mapping framework.

6.2.1 CPU Performance

As mentioned previously, OpenMP directives were used for the CPU benchmark. We annotated parallelizable loops with `omp parallel for` directives, with an additional `collapse` clause for two-dimensional loops. The reduction loops were also annotated with a `reduction` directive. The OpenMP annotated codes were compiled using Intel C++ Composer XE 2013 service pack 1 (ICC). As a baseline reference for comparison, we compiled the original C code without annotations using ICC with the *-O3* and *-parallel* compiler flags. This will in turn enable the *-opt-matmul* flag, which allows the compiler to identify matrix multiplication loop nests if there are any, and replace them with a compiler-generated *matmul* intrinsic. In Table 12, we compare the relative performance of the framework generated code (denoted as FGC-MKL), and the OpenMP compiler generated code (denoted as OpenMP) to the original ICC compiled un-annotated code.

For many of the benchmarks, ICC managed to recognize the *matmul* operations: one such operation in *cg*, three in *gmres*, two in *matexp* and one in *mcl*. However, it is unable to detect any *matmul* operation in *jacobi* or *qr*. For *jacobi*, this was because of additional statements in the loop nest which the compiler failed to abstract. For the *qr* benchmark, the operations involved submatrices which the compiler was not able to detect. When OpenMP directives were added to the benchmarks, the compiler generated threaded code directly from the loops nests. In general, the threaded versions run faster than non-threaded versions. This can be seen from the speedups achieved in Table 12. However, there are also cases such as *matexp* and *mcl* where thread overheads and poor tiling and code generation from the OpenMP compiler resulted in worse performance.

In all benchmarks, the FGC-MKL results showed better performance than the OpenMP compiled results. The main reason is because the BLAS and BLAS-like operations provided by the Intel MKL library were more efficient due to the optimal usage of threading and better memory caching. Although it is easy to write parallel codes using OpenMP, it is difficult to write cache-optimal codes in OpenMP. This is particularly evident in *qr* where OpenMP codes fared worse than the framework, which was able to map operations involving submatrices to optimized library calls.

6.2.2 GPU Performance

Similarly, for the GPU, we compared the performance of the OpenACC compiler generated code and the FGC-CUBLAS codes using with original un-annotated C code as the baseline. The parallelizable loops were annotated with the `acc kernels loop` directive,

with an additional `independent` clause to indicate that there were no loop dependencies, and with the `collapse` clause for two-dimensional loops. Similarly, the reduction loops are also annotated. The OpenACC codes were compiled using the PGI compiler (version 13.10). The GPU codes were executed on the Nvidia Tesla K20 GPU and the results are also presented in Table 12.

In most of the benchmarks, FGC-CUBLAS showed either comparable or better performance compared to the OpenACC compiled codes. This is because FGC-CUBLAS was able to recognize the linear algebraic operations in the code and map them to optimized CUBLAS calls which made better use of architectural features such as software-managed cache. On the other hand, the OpenACC compiled codes were not as optimal. There are several reasons. First, the PGI compiler assigns a thread-block with a default of 128 threads for each kernel. To obtain the optimal number of threads requires extensive tuning of the parameter, which is non-trivial for the user. Second, although OpenACC has a `cache` construct for caching elements or subarrays in the software-managed data cache, this is only useful when the loops are efficiently tiled. Even though adding parallelization directives is a relatively simple task for many users, making effective use the software-managed cache and tuning the parameters is usually much harder for them. Therefore, using our framework is easier as it allows the codes to be mapped to vendor libraries which are already optimized for the underlying hardware.

7 Conclusion

In this paper, we have described a code transformation framework that can recognize and extract code which resembles linear algebraic forms, and map them to optimized vendor libraries. Although users often write code and annotate them using compiler directives to target different hardware platforms, many of such codes contain idioms which have equivalents in highly optimized vendor libraries. Therefore, we proposed an approach to extract known idioms and replace them with optimized library calls. We demonstrated and evaluated the extraction and mapping process from OpenMP and OpenACC annotated codes to Intel MKL and Nvidia CUBLAS, and showed that better performance can be achieved due to the use of these hardware-specific optimized libraries as compared to direct code generation. Another advantage of our approach is that it future-proofs an application because one can quickly target new platforms or use newer optimized libraries as they become available without modifying the original code.

References

[1] I. Buck, "GPU Computing: Programming a Massively Parallel Processor," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2007, pp. 17–17.

[2] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," in *Computer graphics forum*, vol. 26, no. 1, 2007, pp. 80–113.

[3] E. M. Aldrich, "GPU Computing in Economics," in *Handbook of Computational Economics*, ser. Handbook of Computational Economics, K. Schmedders and K. L. Judd, Eds. Elsevier, 2014, vol. 3, pp. 557 – 598.

[4] M. Harris, "Fast Fluid Dynamics Simulation on the GPU," *GPU Gems*, vol. 1, pp. 637–665, 2004.

[5] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig, "Molecular Dynamics Simulations on Commodity GPUs with CUDA," in *High Performance Computing - HiPC 2007*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, vol. 4873, pp. 185–196.

[6] S. Tsutsui and P. Collet, *Massively Parallel Evolutionary Computation on GPGPUs*. Springer-Verlag (New York), 2013.

[7] L. Dagum and R. Menon, "OpenMP: An Industry Standard API for Shared-Memory Programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[8] C. Pheatt, "Intel Threading Building Blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[9] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008.

[10] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering*, vol. 12, no. 3, p. 66, 2010.

[11] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC: First Experiences with Real-world Applications," in *Proceedings of the 18th international conference on Parallel Processing*, ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 859–870.

[12] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A Hybrid Multicore Parallel Programming Environment," in *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[13] *OpenMP Specification for Parallel Programming*. [Online]. Available: http://www.openmp.org/

[14] *GNU Compiler Collection*. [Online]. Available: http://gcc.gnu.org

[15] *IBM XL Compiler*. [Online]. Available: http://www.ibm.com/software/products/en/xlcpp-aix

[16] "Intel C++ Studio XE." [Online]. Available: https://software.intel.com/en-us/intel-parallel-studio-xe/

[17] *OpenACC Directives for Accelerators*. [Online]. Available: http://www.openacc-standard.org

[18] *Cray Programs*. [Online]. Available: http://www.cray.com/Programs.aspx

[19] *PGI Accelerator Compiler (The Portland Group)*. [Online]. Available: http://www.pgroup.com/

[20] *CAPS Enterprise Compiler*. [Online]. Available: http://www.caps-entreprise.com/products/caps-compilers/

[21] *PathScale ENZO Compiler Suite*. [Online]. Available: http://www.pathscale.com/enzo

[22] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "PetaBricks: A Language and Compiler for Algorithmic Choice," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 38–49.

[23] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code Generation for DSP Transforms," *Proceedings of the IEEE, Special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.

[24] K. Brown, A. Sujeeth, H. Lee, T. Rompf, H. Chafi, and K. Olukotun, "A Heterogeneous Parallel Framework for Domain-Specific Languages," in *20th International Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 35–46.

[25] F. G. Van Zee and R. van de Geijn, "BLIS: A Framework for Rapidly Instantiating BLAS Functionality," *ACM Trans. Math. Softw*, 2013.

[26] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammerling, H. Greg, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley, "An

Updated Set of Basic Linear Algebra Subprograms (BLAS)," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151, Jun. 2002.
[27] G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek, "Automating the Generation of Composed Linear Algebra Kernels," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 59.
[28] B. Marker, J. Poulson, D. Batory, and R. Geijn, "Designing Linear Algebra Algorithms by Transformation: Mechanizing the Expert Developer," in *High Performance Computing for Computational Science - VECPAR 2012*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 7851, pp. 362–378.
[29] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA projects," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012037.
[30] C. Ancourt and F. Irigoin, "Scanning Polyhedra with DO Loops," *SIGPLAN Not.*, vol. 26, no. 7, pp. 39–50, Apr. 1991.
[31] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies," *International Journal of Parallel Programming*, vol. 34, no. 3, pp. 261–317, 2006.
[32] P. Feautrier, "Dataflow Analysis of Array and Scalar References," *International Journal of Parallel Programming*, vol. 20, 1991.
[33] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.
[34] *Intel Math Kernel Library Reference Manual*, Accessed 2014.
[35] *Nvidia CUBLAS Reference Manual*, Accessed 2014.

Wen Jun Tan received his B.Eng (Computer) from the Nanyang Technological University (Singapore) in 2011. He is currently pursuing his M.Eng at the same university. His current research interests include: high performance computing, cloud computing, complex adaptive systems and multi-agent systems.

Wai Teng Tang received the bachelor's degree (with honors) from the National University of Singapore. For his postgraduate degree, he worked in the area bioimaging and received the PhD from the same university. He is currently a research scientist at the Institute of High Performance Computing, Agency for Science, Technology and Research, Singapore. His interests are in modeling and simulation, high performance computing, and big data analytics.

Rick Siow Mong Goh is the Director of the Computing Science Department at the A*STAR Institute of High Performance Computing (IHPC). At IHPC, he leads a team of more than 70 scientists in performing scientific research, developing technologies to commercialization, and engaging and collaborating with industry. The research focus areas include high performance computing (HPC), distributed computing, big data analytics, intuitive interaction technologies, and complex systems. His expertise is in discrete event simulation, parallel and distributed computing, and performance optimization and tuning of applications on large-scale computing platforms. Dr. Goh received his Ph.D. in Electrical and Computer Engineering from the National University of Singapore.

Stephen John Turner is Professor of Computer Science in the School of Computer Engineering at Nanyang Technological University (Singapore). He received his MA in Mathematics and Computer Science from Cambridge University (UK) and his MSc and PhD in Computer Science from Manchester University (UK). His current research interests include: high performance computing, cloud computing, parallel and distributed simulation, complex adaptive systems and multi-agent systems.

Weng-Fai Wong received his B.Sc. from the National University of Singapore in 1988, and his Dr.Eng.Sc. from the University of Tsukuba, Japan in 1993. He is now an Associate Professor at the Department of Computer Science at the National University of Singapore. His research interest is in computer architecture, compilers, and high performance computing. He is a member of the ACM, and a Senior Member of the IEEE.