A code transformation framework for linear algebra operations targeting multiple platforms

Tan, Wen Jun

2014

Tan, W. J. (2014). A code transformation framework for linear algebra operations targeting multiple platforms. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/61835>

<https://doi.org/10.32657/10356/61835>

NANYANG
TECHNOLOGICAL
UNIVERSITY

**A CODE TRANSFORMATION FRAMEWORK FOR
LINEAR ALGEBRA OPERATIONS TARGETING
MULTIPLE PLATFORMS**

TAN WEN JUN

SCHOOL OF COMPUTER ENGINEERING

2014

A CODE TRANSFORMATION FRAMEWORK FOR LINEAR ALGEBRA OPERATIONS TARGETING MULTIPLE PLATFORMS

TAN WEN JUN

School of Computer Engineering

A thesis submitted to Nanyang Technological University in partial
fulfilment of the requirement for the degree of Master of Engineering

2014

Acknowledgements

I express my sincere gratitude to my advisor, Prof. Stephen John Turner, for his guidance and encouragement. None of the work described in this thesis would have been possible without his invaluable technical and moral support.

During my involvement in the PSF Project, I have the pleasure of meeting several remarkable people. I owe my colleague Dr Tang Wai Teng a sincere debt of gratitude for his help in this thesis. I am grateful with other members of the PSF Project, Prof Wong Weng-Fai, Ms Wong Yi Wen and Dr Rick Goh Siow Mong, for their insightful advice on the technical aspects of the project.

Finally, I would like to thank my family. I owe my parents and my sisters a huge debt of gratitude for the unconditional support and encouragement they have always given me.

Abstract

Linear algebra is used in many fields of mathematics, natural sciences, computer science, and social science. The researchers in those fields, who are not experienced programmers, may write naïve linear algebra codes for their applications. Despite the availability of multiple hardware platforms for computation, such as vector processing units, multi-core processors and general-purpose graphics processing units, it is particularly difficult for them to optimize their applications for these platforms. It is necessary to make use of hardware-specific architectural features to fully exploit the capabilities of the underlying hardware.

This thesis presents a code transformation framework for assisting users writing naïve linear algebra codes to optimize their applications. The code transformation framework recognizes and extracts the linear algebra operations from naïve codes. The linear algebra intermediate representation can be expressed in terms of a domain specific language. The user can also write linear algebra kernels using this language directly.

Using the intermediate representation, the framework transforms the linear algebra operations to equivalent library calls. There are two possible classes of code generated from the framework: (i) linear algebra template libraries and (ii) BLAS function calls. The linear algebra template libraries present the linear algebra operations in an algebraic form that is easily understandable by the user, and at the same time provides good performance when codes are compiled. It is possible to target multiple platforms through different linear algebra template libraries, such as Eigen and Armadillo. Although C++ template libraries are more intuitive to the user, these template libraries introduce additional overhead due to the library design. Therefore the code transformation framework is extended to generate calls to BLAS for different platforms, such as Intel MKL or Nvidia CUBLAS.

We demonstrate that the code transformation framework can result in better performance in the generated codes compared to those which are generated by existing compilers. This is substantiated by experimental results on different hardware platforms, including vector processing units, multi-core processors and graphics processing units.

Contents

1	Introduction	1
1.1	Motivating Example	2
1.1.1	Naïve Implementation	2
1.1.2	The Need for Abstraction	3
1.2	Objectives	4
1.3	Organization of the Thesis	5
2	Background	7
2.1	Vector Processing Units	8
2.2	Multi-core Processors	9
2.2.1	OpenMP	10
2.3	Graphics Processing Units (GPU)	11
2.3.1	GPU Architecture	11
2.3.2	CUDA Programming Model	12
2.3.3	Directives for GPU Programming	13
2.4	Basic Linear Algebra Subprograms	15
2.4.1	Vendor-Optimized BLAS	15
2.4.2	BLAS Programming Model	16
2.5	C++ Template Libraries for Linear Algebra	17
2.5.1	Eigen C++ Template Library for Linear Algebra	18
2.5.2	Armadillo C++ Linear Algebra Library	21
2.6	Summary	23
3	A Code Transformation Framework	25
3.1	Overview of Framework	25
3.2	Linear Algebra Operations	28
3.3	Linear Algebra Language	29
3.4	Summary	30
4	Extraction Engine	31
4.1	Array Size Annotation	32
4.2	Domain and Shape Recognition for Arrays	33
4.2.1	Loop Extraction	33
4.2.2	User Annotated Loops	37
4.2.3	Polyhedral-based Extraction	39
4.3	Canonical Reduction	41
4.4	Linear Algebra Recognition	41
4.5	Summary	44

5	Transformation Engine	45
5.1	Linear Algebra Template Libraries	46
5.1.1	Vector Processing Unit	46
5.1.2	Multi-core Processor	47
5.1.3	Graphics Processing Unit	47
5.1.4	Adaptive Mapping	48
5.2	Direct to BLAS	50
5.2.1	Statement Fusion	52
5.2.2	Operation Mapping	53
5.2.3	Platform-Specific Code Generation	56
5.3	Summary	59
6	Evaluation and Results	61
6.1	Linear Algebra Template Libraries	62
6.1.1	Setup	62
6.1.2	Evaluation	65
6.1.3	Adaptive Mapping	66
6.2	Direct to BLAS	70
6.2.1	Setup	70
6.2.2	Performance Comparison	72
6.3	Summary	74
7	Comparisons with Related Work	75
7.1	Source-to-source Transformation	75
7.2	Adaptive Code Generation	76
7.3	Optimization	78
7.4	Alternatives to BLAS	79
7.5	Summary	80
8	Conclusion	81
8.1	Summary	81
8.2	Future Work	83
A	Linear Algebra Language	85
A.1	Types	86
A.2	Expressions	87
A.2.1	Literals	87
A.2.2	Array Expression	88
A.2.3	N-ary Expression	88
A.2.4	Unary Expression	89
A.2.5	Order of Operation	90
A.3	Statements	91
A.3.1	Assignment Statement	91
A.3.2	Declaration Statement	92
A.3.3	Conditional Statement	92
A.3.4	Statement Block	92
A.3.5	Loops	93
A.4	Function	94

B Additional Results	95
References	99

Chapter 1

Introduction

In this modern era of computing on different hardware platforms, application developers are increasingly turning to parallelism to achieve higher application performance. These modern computing systems, for example a workstation comprising of a multi-core central processing unit (CPU) and a graphics processing unit (GPU), expose many levels of hardware parallelism to the programmer, which can range from vector processing units in each processor core, to multi-threading on multi-core processors, as well as general-purpose computing on GPU [1, 2]. To make effective use of these resources, the programmer usually has to have a deep understanding of the different hardware architectures in order to write parallel code that targets them.

In order to retrofit code with parallelism to take advantage of the new hardware architectures, a common way to parallelize the program is to make use of well-tuned programming libraries such as the Intel Math Kernel Library (MKL) [3], Eigen C++ Template Library [4], Nvidia CUBLAS Library [5], and replace matching code fragments with functions offered by these libraries. However, coding a program with a specific target library has many disadvantages. One problem is the issue of maintainability and portability – as hardware architectures evolve, programs will have to be constantly updated to make use of new programming interfaces for the new architectures, which quickly becomes a burden to the maintainers. For example, code that uses the Intel MKL to target multi-core processors will have to be rewritten and additional architectural changes may have to be made to the code in order to

target the GPU.

Apart from the tediousness of having to port code to new architectures when they become available, another problem concerns that of performance scalability. In many programs, performance libraries such as Intel MKL or AMD Core Math Library (ACML) [6] are used for computations. However, this restricts the parallelism of the application to only vectorization and threading. In terms of performance scalability, this is not ideal since for large problem sizes, many data parallel operations can be more effectively processed on the GPU.

1.1 Motivating Example

1.1.1 Naïve Implementation

The example illustrates a typical case where programmers explicitly express the computations to be done using *for*-loops without resorting to tuned libraries. We shall call this a *naïve implementation* of an algorithm. Figure 1.1 shows a piece of code that is written in this fashion. It shows one of the most natural ways the user can write to express the computation. The code is simple and can be easily ported to any platform with a working C compiler. However, there are no guarantees on the performance of the code.

Even though the user may expect the compiler to generate an optimized form of the code, in practice, the result is very much compiler-dependent. For example, suppose the code in Figure 1.1 is compiled using the Intel C++ Compiler XE 2013 (ICC) with the auto-parallelizer enabled. However, no loops are parallelized with threading as n is not assigned a specific number; ICC is unable to determine the loop bounds. Although ICC can recognize a matrix multiply operation and substitute it with a compiler intrinsic, it fails to recognize the matrix multiply at line 7 as there are other operations at lines 5 and 9. Therefore to enable the compiler to perform the optimization, it requires the user to extensively change the way the code is written.

```

1  for (i = 0; i < n; i++){
2      x_old[i] = x[i];
3  }
4  for (i = 0; i < n; i++){
5      x[i] = 0.0;
6      for (j = 0; j < n; j++){
7          x[i] += T[i][j] * x_old[j];
8      }
9      x[i] += c[i];
10 }

```

Figure 1.1: Code fragment that is written naïvely.

Size of data	Run Time (s)		Speedup (> 1)
	Sequential	Intel MKL	Slowdown (< 1)
4096	5.79	12.68	0.46
8192	22.82	25.15	0.91
16384	90.59	66	1.37
32768	365.08	216.08	1.69

Table 1.1: Performance of sequential and Intel MKL versions of Figure 1.1 as the size of data increases.

Alternatively, programmers may choose to make use of optimized or tuned third-party libraries for their computation needs. The example from Figure 1.1 is rewritten using Basic Linear Algebra Subprograms (BLAS) from Intel MKL. BLAS is a common standard Application Programming Interface (API) for linear algebra operations. The code is linked with the multi-threaded library of Intel MKL.

The results in Table 1.1 compare the performance of the serial version of the code fragment from Figure 1.1 compiled with ICC, versus the BLAS version. When the problem size is small, the overhead of using Intel MKL is high due to the startup cost of threading. However, performance can be gained by executing the BLAS version when the problem size exceeds 16384 elements.

1.1.2 The Need for Abstraction

From the above example, we have shown two common ways of writing a program, either the naïve way or by using optimized libraries. The goal of the framework is to extract linear algebra operations from the existing code, and then map them to

$$\begin{aligned}
x_{old} &\leftarrow x \\
x &\leftarrow 0_n \\
x &\leftarrow T * x_{old} + x \\
x &\leftarrow c + x
\end{aligned}$$

Figure 1.2: Linear algebra representation of the code in Figure 1.1

optimized libraries for different platforms, such as Intel MKL or Nvidia CUBLAS.

As an illustration, linear algebra operations can be extracted from the code in Figure 1.1 through pattern recognition and internally represented by the linear algebra operations in Figure 1.2.

1.2 Objectives

To address many of the above issues, we propose a code transformation framework based on extraction of linear algebra operations, and generate code for different hardware architectures. The goal of the framework is to help programmers to transform their code and map them to different hardware that is suitable for the underlying computations. By using *linear algebra operations*, it is possible to apply more aggressive static and dynamic optimizations according to the additional mathematical properties of the operations. These optimizations will be difficult or impossible on the original codes due to insufficient information on the data structures and algorithms. Compilers will not perform certain optimizations in order to maintain the correctness of the program output.

The objectives of the framework include:

- Extracting linear algebra operations from existing code
- Performing optimizations on the linear algebra operations
- Generating code that targets different platforms

so as to achieve scalable performance when the code is executed on the hardware.

This framework differs from many other frameworks in that the focus is on the *linear algebra operations* as a way to achieve execution on multiple platforms.

As such, the main contributions of this thesis are as follows.

- We present a code transformation framework which transform from naïve C/C++ source code to linear algebra codes targeting different hardware platforms.
- We propose the procedures to identify linear algebra operations from the original source code using pattern-matching rules on the abstract syntax tree.
- The linear algebra operations extracted can be expressed in terms of a domain-specific language we have designed.
- We describe two different components that can be used to generate different codes from the linear algebra operations: linear algebra template libraries or BLAS libraries directly.
- Experimental evaluation has been performed using a set of scientific applications that utilize linear algebra operations. Experiments reveal that through this unified transformation framework, codes can be easily transformed to target different platforms using the optimized linear algebra libraries.

Part of this work has been published in IEEE Transactions on Parallel and Distributed Systems (TPDS) [7] and pre-prints are already available.

1.3 Organization of the Thesis

The remainder of the thesis is organized as follows:

Chapter 2 introduces the background information for this framework.

Chapter 3 gives an architectural overview of the framework, overview of the linear algebra operations supported and the Linear Algebra Language used to express the linear algebra operations.

Chapter 4 describes the *Extraction Engine* and processes to extract linear algebra operations from the naïve codes.

Chapter 5 presents the *Transformation Engine*, which performs code transformation to map the linear algebra operations to respective library calls.

Chapter 6 evaluates the effectiveness of the code generation framework by experiments on multiple platforms.

Chapter 7 compares some of the existing source-to-source transformation frameworks and the difference of these framework with our current design. It also examines the area of adaptive code generation and the optimizations on array languages.

Chapter 8 concludes the thesis and discusses some of the future works for this framework.

In Appendix A, the language specification for the Linear Algebra Language is explained in detail. Figures illustrating the additional results from Chapter 7 are shown in Appendix B.

Chapter 2

Background

Modern computing hardware comprises of multiple different platforms within the same computing system. A brief overview of the hardware architecture and programming model for different platforms is given in this chapter:

1. Vector processing units (Section 2.1),
2. Multi-core processors (Section 2.2), and
3. General purpose graphics processing unit (Section 2.3).

In Section 2.4, the commonly-used linear algebra software, Basic Linear Algebra Subprograms (BLAS), is presented. A brief overview of the BLAS API is shown.

Section 2.5 introduces the use of linear algebra libraries that encapsulate the functionalities of BLAS. They feature intuitive syntax for the linear algebra operations, making it easy for the user to understand. Two linear algebra libraries based on C++ template libraries are further described in detail: Eigen C++ Template Library for Linear Algebra (Section 2.5.1) and Armadillo C++ Linear Algebra Library (Section 2.5.2). Similarly, the corresponding APIs for Eigen and Armadillo are listed and code fragments of these libraries are shown as examples.

Figure 2.1: SIMD unit executes a single instruction on an array of data.

2.1 Vector Processing Units

A vector processing unit implements an instruction set containing instructions that operate on one-dimensional arrays of data called vectors. Today, most commodity CPUs implement architectures that feature instructions for vector processing, typically known as SIMD (Single Instruction, Multiple Data). SIMD describes the short-vector processing units that perform the same operation on multiple elements in the vector simultaneously. Figure 2.1 shows the SIMD unit executing a single instruction on an array of data. SIMD may have restrictions on data alignment to be efficient. When the data is not aligned to the vector width, additional overhead of shifting the data in the register to achieve data alignment is required. Gathering data into SIMD registers and scattering it to the correct destination locations also requires additional transfer overhead.

Streaming SIMD Extensions (SSE) is an SIMD instruction set extension to the Intel architecture. Different versions of SSE support from 64-bit short vectors to 128-bits. The newer extensions are called Advanced Vector Extensions (AVX), and support 256-bits to 512-bit short vectors. These extensions can process vectors on different data-types, including 8-bits to 64-bits integers and 32-bits or 64-bits

```

1  v3.x = v1.x + v2.x;
2  v3.y = v1.y + v2.y;
3  v3.z = v1.z + v2.z;
4  v3.w = v1.w + v2.w;

```

(a) Addition of two vectors ($v1$ and $v2$) of four 32-bit single precision numbers assigning the results to vector ($v3$).

```

1  v3 = __builtin_ia32_addps(v1, v2);

```

(b) Pseudo-code of addition of two vectors ($v1$ and $v2$) using a single intrinsic function (*addps*) assigning the results to vector ($v3$).

Figure 2.2: Example of demonstrating the use of SIMD.

floating point numbers. Other than the typical SIMD data movement and arithmetic operations supported in SSE, AVX adds support for three-operand operations, such as the fused multiply-accumulate which perform multiplication and addition on three operands using a single instruction.

To utilize SIMD in the program, the user can use intrinsic functions, which are inbuilt functions in C, that are mapped directly to the SIMD instructions. An example of using intrinsic functions is shown in Figure 2.2. Otherwise, the user can apply vectorization using compilers which converts a loop, that is operating on each element of an array, to a loop operating on a partition of the array where each partition is of the vector width. However, loop dependence analysis is required to ensure there are no data flow dependencies between iterations of statements within the loop. Violation of this check will result in incorrect results when SIMD instructions are used.

2.2 Multi-core Processors

Originally CPUs were developed with only one core on a physical chipset. As the rate of clock speed improvements slowed, increased use of parallel computing in the form of multi-core processors has been pursued to improve overall processing performance.

Figure 2.3: Each thread maps to a single core on the multi-core processor.

The multi-core processors feature multiple cores on the same physical chipset. The proximity of multiple CPU cores on the same die reduces the distance the signals need to travel between the cores, which significantly improves the performance of the shared cache and decreases the power required to drive the cores. This also allows multi-threaded applications to map each thread to a different core on the CPU. Figure 2.3 illustrates the mapping of a thread to each physical core on the CPU.

2.2.1 OpenMP

The OpenMP [8] 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 [9], IBM [10], and Intel [11].

By using OpenMP, existing codes may be ported to multi-core processors with less effort by users. OpenMP allows programmers to simply add annotations in the form of compiler directives to regions of codes in order to parallelize them for execution on multi-core processors. In the annotated code section, the main program

```

1 #pragma omp parallel for
2 for (int i = 0; i < N; i++)
3 {
4     v3[i] = v1[i] + v2[i];
5 }

```

Figure 2.4: An code fragment of element-wise addition of two vectors ($v1$ and $v2$) of N elements.

thread will fork a number of slave threads to distribute the task between them. In many cases, there is no need to modify the data structures or make algorithmic changes to the underlying code base when adding the directives.

An example of an OpenMP annotation is shown in Figure 2.4. The pragma `omp parallel` is used to fork additional threads to carry out the work enclosed in the construct in parallel. Using the `parallel` pragma with the `for` pragma, it can split the loop iterations between the threads.

2.3 Graphics Processing Units (GPU)

Users may program general-purpose applications to be executed on their GPUs using the Compute Unified Device Architecture (CUDA), which is a parallel computing platform and programming model for Nvidia GPUs. Nvidia GPUs have a parallel throughput architecture that executes many concurrent threads using the same instruction.

2.3.1 GPU Architecture

The Kepler architecture [12] was released in 2012. It utilizes a unified clock to achieve better efficiency. A full Kepler GK110 implementation includes 15 new Generation Streaming Multiprocessors (SMX) and 6 memory controllers, with 192 cores per SMX. While each SMX has the same amount of registers (32 KB) as the Streaming Multiprocessors (SM) of the Fermi architecture, the maximum number of registers per thread is increased to 255. This aims to reduce register pressure faced in the Fermi, where performance is affected when a variable is moved from the

	Geforce 680	Tesla K20
Type of GPU	GK104	GK110
Peak single precision floating point performance (Tflops)	3.09	3.52
Memory bandwidth (GB/sec)	192	208
Memory size GDDR5 (GB)	2	5
Number of SMX	8	13
CUDA cores	1536	2496
Clock (MHz)	1006	706
Compute Capability	3.0	3.5

Table 2.1: Features of Nvidia Kepler Architecture: Geforce 680 and Tesla K20.

register to the Level 1 cache. Even though the clock speed is lower, Kepler is able to achieve a similar or better level of performance by increasing the number of cores per SMX. Kepler also increases the number of schedulers to 4.

An overview of the features of the Geforce 680 and Tesla K20 (Kepler) is shown in Table 2.1.

2.3.2 CUDA Programming Model

CUDA extends the C++ programming language with data parallel constructs to support execution on GPU. Figure 2.5 shows the thread hierarchy for CUDA. Each thread executes a single instance of the CUDA kernel, with per-thread registers. A thread block groups the threads and concurrently execute them with the same CUDA kernel. The geometry and size of the thread block can be defined at the kernel launch, where CUDA supports up to three dimensional thread blocks. All the threads within the thread block have access to the same shared memory. The threads in the block can cooperate among themselves through barrier synchronization. A grid executes the CUDA kernel with a set of thread blocks. The grid can be defined with up to two dimensions. All the threads in the grid have access to the global memory region on the GPU allocated by the program.

CUDA is designed to map the logical threads to the physical cores on the GPU.

1. A GPU executes one or more kernel grids.

Figure 2.5: CUDA Thread Hierarchy

2. A SMX executes one or more thread blocks.
3. A SMX executes a warp of 32 threads.

A warp refers to the group of threads that the GPU scheduler executes together.

When a CUDA program executes, the main program is running on the host system, while the CUDA kernels are executed on the GPU. CUDA also provides an API to allocate memory on the GPU and transfer data between the host memory and the GPU memory. To write a CUDA program, the user needs to identify parallelizable loops and rewrite them as CUDA kernels. An example of writing a CUDA program is shown in Figure 2.6. The vectors $v1_{gpu}$, $v2_{gpu}$ and $v3_{gpu}$ are allocated on the GPU using `cudaMalloc`. The two input vectors, $v1$ and $v2$ are copied into the GPU memory using `cudaMemcpy`. `kernel<<<1, N>>>` executes the kernel with 1 thread block and N threads per block. After the execution of the kernel, the resultant vector $v3$ is copied out of the GPU memory. At the end of the application, the memory allocated on the GPU is released by calling `cudaFree`.

2.3.3 Directives for GPU Programming

To write optimal programs to execute on the GPU requires the user to have in-depth knowledge of both the CUDA programming model and the Nvidia GPU architecture. Therefore the OpenACC [13] standard is defined by a group comprising of major computer vendors for targeting hardware accelerators such as GPUs, and is supported by compilers from Cray [14], PGI [15], and CAPS [16]. Like OpenACC,

```

1  __global__ void kernel(int* v1, int* v2, int* v3){
2      v3[i] = v1[i] + v2[i];
3  }
4
5  //Host Code
6  int* v1_gpu, v2_gpu, v3_gpu;
7
8  cudaMalloc(&v1_gpu, N*sizeof(int));
9  cudaMalloc(&v2_gpu, N*sizeof(int));
10 cudaMalloc(&v3_gpu, N*sizeof(int));
11
12 cudaMemcpy(v1_gpu, v1, N*sizeof(int), cudaMemcpyHostToDevice);
13 cudaMemcpy(v2_gpu, v2, N*sizeof(int), cudaMemcpyHostToDevice);
14
15 kernel<<<1, N>>>(v1_gpu, v2_gpu, v3_gpu);
16
17 cudaMemcpy(v3, v3_gpu, N*sizeof(int), cudaMemcpyDeviceToHost);
18
19 cudaFree(v1_gpu);
20 cudaFree(v2_gpu);
21 cudaFree(v3_gpu);

```

Figure 2.6: An example of CUDA program based on Figure 2.4.

OpenHMPP [17] can be used to target GPUs, but is currently only supported by compilers from CAPS and PathScale [18]. By using OpenACC, existing codes may be ported to 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 GPUs. In many cases, there is no need to modify the data structures or make algorithmic changes to the underlying code base when adding the directives.

An example demonstrating an OpenACC program is shown in Figure 2.7. The user identifies a block of code as having parallelism and the compiler generates a parallel GPU kernel for that loop. The compiler also automatically decides the configuration for the grid and thread block based on the number of loop nests. The user can also add annotations for explicit data allocation and transfer: `create`, `copyin`, `copyout`, `copy` (copy in and out).

```

1 #pragma acc parallel loop
2 for (int i = 0; i < N; i++)
3 {
4     v3[i] = v1[i] + v2[i];
5 }

```

Figure 2.7: OpenACC code based on Figure 2.4.

2.4 Basic Linear Algebra Subprograms

Developers of applications involving linear algebra operations often implement certain low level operations as subprograms or subroutines, such as dot product or matrix-vector product. This results in the creation of basic building blocks of fundamental linear algebra operations that are commonly used. Since the majority of execution time is spent on these fundamental operations, optimizing these routines can effectively reduce the total execution time of the application.

These efforts lead to the development of libraries designed for linear algebra. The most widely known are the Basic Linear Algebra Subroutines (BLAS) and Linear Algebra PACKage (LAPACK) [19] libraries. The programming interfaces for these libraries are fairly standardized, with many vendors providing their own high-performance drop-in replacement libraries.

2.4.1 Vendor-Optimized BLAS

Intel Math Kernel Library (MKL) [3] provides a vectorized and threaded math library for Intel processors. Standard APIs for BLAS and LAPACK libraries are also implemented as part of Intel MKL. Intel MKL also provides the flexibility to meet the user's requirements. When the user compiles his application with Intel MKL, he has the option to select a single-threaded library or multi-threaded library. The user can enable the flag to obtain reproducible results for his application requirement, at the cost of performance.

Nvidia CUDA Basic Linear Algebra Subroutines (CUBLAS) [5] is a vendor-optimized BLAS library for Nvidia GPU. To use CUBLAS, GPU initialization is

Level 1		
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
Level 2		
Matrix Vector Product	$y \leftarrow \alpha A \times x + \beta y$	GEMV
Outer Product	$A \leftarrow \alpha x \otimes y + \beta A$	GER
Level 3		
Matrix Matrix Product	$C \leftarrow \alpha A \times B + \beta C$	GEMM

Table 2.2: Some of the BLAS Operations on General Matrices

required prior to invocation of CUBLAS routines, as well as clean-up at the end of the application life-cycle. The user needs to allocate the memory and transfer the matrices or vectors in the computation to the GPU. After execution of the CUBLAS operations, the results are transferred back to the host. For compatibility with the original BLAS, CUBLAS uses column-major storage for the matrices on the GPU. Due to the additional initialization and termination routines for the GPU, and the memory transfers between the host and the GPU, users cannot easily port their applications to utilize CUBLAS. Users need to minimize the amount of data transfer between the host and GPU to achieve better performance.

2.4.2 BLAS Programming Model

Specifications for Level 1 BLAS are defined for scalar and vector operations. Additional specifications are added for Level 2 and 3 BLAS, concerned with matrix-vector and matrix-matrix operations respectively. BLAS supports four data types: (i) single-precision real number, (ii) double-precision real number, (iii) single-precision complex number, and (iv) double-precision complex number. It also supports different types of matrices, e.g. general dense matrix, Hermitian matrix, symmetric matrix and triangular matrix, with different storage formats. Examples of different levels of BLAS operations are shown in Table 2.2.

Figure 2.8 shows an example of an application written in BLAS based on C

```

1  scopy(n,x,1,x_old,1);
2  sscal(nsize,0,x,1);
3  sgemv(CblasRowMajor,CblasNoTrans,n,n,1,T,n,x_old,1,0,x,1);
4  saxpy(n,1,c,1,x,1);

```

Figure 2.8: C code fragment based on Figure 1.1 using BLAS.

code from Figure 1.1. From Figure 2.8, we can see that the application can be decomposed into several fundamental BLAS operations. Although we can easily build portable linear algebra applications using BLAS as the basic building blocks, their programming interfaces requires a number of parameters. Each BLAS call requires the properties of the arrays to be passed in as parameters, such as storage format of the matrix (row or column format), the dimension of the matrix, the offset of the matrix, and the pointer to the matrix. In addition, the user also needs to define options such as whether transposition is applied to the matrix, and the scaling factor of each matrix. The verbosity of their programming interfaces makes them quite unwieldy and error-prone in day-to-day use.

2.5 C++ Template Libraries for Linear Algebra

To improve on the ease and usability of BLAS, several C++ Linear Algebra libraries have been developed to encapsulate the functionalities of BLAS. The libraries implement matrix or vector classes with overloaded operators. These make the linear algebra operations intuitive to program using vector or matrix objects with regular mathematical operators. Some of these libraries make extensive use of C++ template meta-programming [20]. C++ template meta-programming is used by the compiler to generate temporary source code that is merged by the compiler with the rest of the code. Using the templates, the compiler can generate specialized code for different numeric types and for fixed-size arrays, allowing type-specific or size-specific optimizations, such as loop-unrolling to be applied.

These template libraries also implement a delayed-operation framework (also known as *lazy evaluation*) that allows the combination of several operations into one

and hence can considerably reduce the generation of temporary objects. This in turn can provide considerable performance improvements. However, most libraries apply lazy evaluation only to sub-expressions, and not across statements. In addition, lazy evaluation can be incorrect in cases where the intermediates need to be evaluated, or can be inefficient in cases where caching of the intermediates gives a better performance.

BLAS operations often consist of a major operation with a scaling term and addition term, e.g. $c = \alpha a \bullet b + \beta c$. Most libraries evaluate the matrix product $a \bullet b$ to an intermediate first. It could be more efficient to pass all the terms to BLAS, instead of evaluating the matrix product and addition separately.

2.5.1 Eigen C++ Template Library for Linear Algebra

Eigen C++ Template Library [4] is designed for vector and matrix operations, and supports all standard numeric types, such as integer, single-precision and double-precision floating-point numbers. Some explicit optimizations have been implemented in the library, such as loop unrolling, vectorization and tiling. The library will perform loop-unrolling for small fixed-size matrices if the loop-count is known at compile-time. It generates explicit vectorization for different instruction sets, and falls back to non-vectorized code if there are no equivalent instructions. The operations are also tiled according to cache size. In addition, lazy evaluation is implemented to reduce the amount of intermediate copying required. In the newer version 3.1, Eigen added optional support for Intel MKL, where some matrix-matrix and matrix-vector floating-point operations such as matrix-matrix multiplication can be executed using Intel MKL.

Lazy Evaluation

In most cases, lazy evaluation is performed by Eigen. There are three circumstances in which *Eigen* chooses immediate evaluation instead of lazy evaluation.

The first case is when it sees an assignment $\mathbf{a} = \mathbf{b}$; and the expression $\mathbf{b}$ has the

```
1 matrix = matrix * matrix;
```

(a) The matrix multiplication of `matrix` assigning to the same matrix.

```
1 matrix1 = matrix2 + matrix3 * matrix4;
```

(b) The product `matrix3 * matrix4` is evaluated immediately into a temporary matrix.

```
1 matrix1 = matrix2 * (matrix3 + matrix4);
```

(c) Each coefficient of the expression `matrix3 + matrix4` is going to be used several times in the matrix product.

Figure 2.9: Examples of immediate evaluation performed by Eigen.

evaluate-before-assigning flag. The most important example of such an expression is the matrix product expression, shown in Figure 2.9a. *Eigen* first evaluates `matrix * matrix` into a temporary matrix, and then copies it into the original matrix. The cost of the matrix copy is negligible compared to the cost of the matrix product itself.

The second circumstance in which *Eigen* chooses immediate evaluation is when it sees a nested expression such as `a + b` where `b` is already an expression having the evaluate-before-nesting flag. Again, the most important example of such an expression is the matrix product expression, as shown in Figure 2.9b. Indeed, experiments showed that it is often beneficial for performance to evaluate matrix products immediately when they are nested into larger expressions.

The third case in which Eigen chooses immediate evaluation, is when its cost model shows that the total cost of an operation is reduced if a sub-expression gets evaluated into a temporary. In certain cases, an intermediate result is sufficiently costly to compute and is reused sufficiently many times. For example in Figure 2.9c, instead of computing the sum, `matrix3 + matrix4`, every time, it is more efficient to compute it once and store it in a temporary variable. Therefore Eigen evaluates `matrix3 + matrix4` into a temporary variable before evaluating the product.

Static Integer Vector	<code>Matrix<int, n, 1> v;</code>
Dynamic Integer Vector	<code>Matrix<int, Dynamic, 1> v(n,0);</code>
Static Float Matrix	<code>Matrix<float, n, n> m;</code>
Dynamic Float Matrix	<code>Matrix<float, Dynamic, Dynamic> m(n,n);</code>

Table 2.3: Eigen object declarations with different data types, where n is the size of the vector/matrix.

Operations	Code Examples
Addition	<code>mat3 = mat1 + mat2; mat3 += mat1;</code>
Scalar Product	<code>mat3 = mat1 * s1; mat3 *= s1; mat3 = s1 * mat1;</code>
Vector Product (Column)	<code>col2 = mat1 * col1;</code>
Vector Product (Row)	<code>row2 = row1 * mat1; row1 *= mat1;</code>
Matrix Product	<code>mat3 = mat1 * mat2; mat3 *= mat1;</code>
Transposition	<code>mat1 = mat2.transpose();</code>
Dot Product	<code>scalar = vec1.dot(vec2);</code>
Outer Product	<code>mat = col1 * col2.transpose();</code>
Cross Product	<code>vec3 = vec1.cross(vec2);</code>

Table 2.4: Code examples of Linear Algebra Operations in Eigen. `s` refers to a scalar variable, `vec` to a vector and `mat` to a matrix.

Eigen Programming Model

Eigen supports both vector and matrix objects. The vector objects are matrix objects with only one column. If the size of the matrix is known, the Eigen objects can be declared with the size provided in the arguments of the template instantiation, which will allocate the object on the stack. Otherwise, the Eigen objects are declared with dynamic size, and allocated on the heap instead. Examples of the declarations are shown in Table 2.3.

The Eigen API is extremely clean and expressive while feeling natural to C++ programmers, due to expression templates. Implementing an algorithm on top of Eigen is similar to writing pseudo-code for linear algebra operations. A list of linear operations expressed in Eigen is shown in Table 2.4. To demonstrate the expressiveness of Eigen, Figure 2.10 show the code based on Figure 1.1, written for Eigen.

```

1 x_old = x;
2 x.setZero();
3 x += T * x_old;
4 x += c;

```

Figure 2.10: Code fragment based on Figure 1.1 using Eigen.

2.5.2 Armadillo C++ Linear Algebra Library

Similar to Eigen, Armadillo C++ linear algebra library [21] consists of generic vector and matrix classes. Armadillo is able to make extensive use of existing libraries such as vendor-tuned BLAS libraries to achieve increased performance and speed. Internally, Armadillo does not support multi-core processors. However, by using multi-threaded vendor-tuned implementations of BLAS, Armadillo is able to make full use of multi-core processors for computation. The main performance of Armadillo depends on the underlying implementation of BLAS to which it is linked. Armadillo also uses template meta-programming as part of a mechanism for implementing a delayed evaluation approach for evaluating mathematical expressions. This approach combines several operations into one, in order to reduce (or eliminate) the need for compiler-generated temporary objects, as well as to work around structural limitations of the class construct within the current version of the C++ language.

Lazy Evaluation

All element-wise operations (such as addition, scaling, etc.) can be combined. Armadillo also has several other intelligent expression evaluators, including: (i) optimization of the order of matrix multiplication, with a view to reduce the amount of time and memory taken (accomplished by reducing the size of the required temporary matrices); (ii) detecting the 1×1 matrix as scalar and apply scaling operations.

Static Integer Vector	<code>Col<int>::fixed<n> v;</code>
Dynamic Integer Vector	<code>Col<int> v(n);</code>
Static Float Matrix	<code>Mat<float>::fixed<n,n> m;</code>
Dynamic Float Matrix	<code>Mat<float> m(n,n);</code>
Static Double Cube	<code>Cube<double>::fixed<n,n,n> c;</code>
Dynamic Double Cube	<code>Cube<double> c(n,n,n);</code>

Table 2.5: Armadillo object declarations, where n is the size of the vector/matrix.

Operations	Code Examples
Addition	<code>mat3 = mat1 + mat2; mat3 += mat1;</code>
Scalar Product	<code>mat3 = mat1 * s1; mat3 *= s1; mat3 = s1 * mat1;</code>
Vector Product (Column)	<code>col2 = mat1 * col1;</code>
Vector Product (Row)	<code>row2 = row1 * mat1; row1 *= mat1;</code>
Matrix Product	<code>mat3 = mat1 * mat2; mat3 *= mat1;</code>
Transposition	<code>mat1 = trans(mat2);</code>
Dot Product	<code>scalar = dot(vec1, vec2);</code>
Outer Product	<code>mat = col1 * trans(col2);</code>
Cross Product	<code>vec3 = cross(vec1, vec2);</code>

Table 2.6: Code examples of Linear Algebra Operations in Armadillo. **s** refers to a scalar variable, **vec** to a vector and **mat** to a matrix.

Armadillo Programming Model

Armadillo is designed to have a similar syntax (API) to Matlab. It provides efficient classes for vectors, matrices and cubes. Similar to Eigen, fixed size objects are also declared through the template arguments, which results in static allocation of the objects. Examples of the declarations are shown in Table 2.5. Most operations are also mapped to overloaded operators, member functions or function calls in Armadillo. Matrix multiplication is mapped to the `*` arithmetic operator while the transposition operation uses a function call `trans(B)`. Table 2.6 show a list of linear operations expressed in Armadillo. An example of the code from Figure 1.1 written in Armadillo is shown in Figure 2.11.

```

1 x_old = x;
2 x.zeros();
3 x += T * x_old;
4 x += c;
```

Figure 2.11: Code fragment based on Figure 1.1 using Armadillo.

2.6 Summary

This chapter gives an overview of the multiple hardware platforms (vector processing units, multi-core processors and GPUs) and some of the programming models for these platforms (vector intrinsic functions, OpenMP, CUDA and OpenACC). The industry standard BLAS library is introduced with the vendor-optimized libraries, Intel MKL and Nvidia CUBLAS, for different hardware platforms. Two C++ template libraries for linear algebra operations, Eigen and Armadillo, are described. Examples are given to show that these template libraries can express the linear algebra operations in a way that allows the users to understand the algorithm easily.

Chapter 3

A Code Transformation Framework

3.1 Overview of Framework

The architecture of the framework is shown in Figure 3.1. The C/C++ parser first reads in the given source code and constructs an abstract syntax tree representation of the code. The abstract syntax tree is manipulated by the *Extraction Engine* to extract the linear algebra operations. The engine performs the following steps. First, based on user annotations, the engine provides additional information regarding the size and dimension of the arrays. Next, the critical domain information from the abstract syntax tree is extracted. The extraction comprises of three alternative components:

- Loop extraction through loop transformation,
- Extraction of user annotated loops, and
- Extraction using a polyhedral model of the loops.

After the domain information is extracted, the AST is normalized using the canonical reduction. Finally, pattern recognition is performed on the linear algebra operations. The intermediate representation of the linear algebra operations in the abstract

Figure 3.1: Architecture of the framework.

syntax tree can be expressed using a domain-specific language for linear algebra, Linear Algebra Language (LAL). The language specification of LAL can be found in Appendix A. Other than extracting from the source codes, the user can also write the code directly using the LAL.

The *Transformation Engine* will then decide the code to generate according to the selected hardware platform. The selected code generator is invoked and outputs the transformed code for the target platform. Currently, the transformation engine supports two types of code to be generated: (i) Linear Algebra Template Libraries, and (ii) BLAS function calls. For the linear algebra template libraries, there are three platform targets for code generation: the vector processing unit, multi-core processor and GPU. Targeting the different platforms, our framework generates Eigen (vector processing unit), Armadillo-MKL (multi-core processor) and Armadillo-CUBLAS (GPU). For BLAS function calls, we generate vendor-optimized BLAS libraries according to the platform targets: Intel MKL (vector processing unit and multi-core processor) and Nvidia CUBLAS (GPU). Finally, the transformed code can be compiled and executed on the targeted hardware.

The benefits of the framework are as follows.

1. It allows code to be written in a naïve way, which can then be executed on different hardware architectures. Since we are targeting the C/C++ language, it allows the user to mix linear algebra operations with their current code-base without a need for a Foreign Function Interface (such as `ctypes` in Python). It also allows existing code, which is written sequentially, to be retrofitted with different levels of parallelism with less effort required.
2. It enforces separation of concerns between writing idiomatic code and writing optimized code. The user can focus on the task of developing the algorithm without worrying about the performance because optimization is deferred to the code generation process.
3. The framework has a modular architecture. As hardware evolves, and as soft-

ware interfaces change, components that target the hardware can be developed and added into the framework independently of the user's source code. This reduces the total cost of maintaining and upgrading the user's source code.

4. It enables the user to target multiple hardware platforms through a single consistent interface.

3.2 Linear Algebra Operations

This section describes the linear algebra operations supported in the framework.

A scalar is denoted by s , a vector by T_i , a matrix by T_{ij} , where the number of subscripts denote the order, or dimension, of the arrays. The corresponding arrays they represent are $T[i]$ and $T[i][j]$ respectively. Note that the notation T_{ij} is different from T_{ji} in that their subscript indices are interchanged. This represents a permutation of the storage order. In array form, they are given by $T[i][j]$ and $T[j][i]$, respectively.

The following is a list of the fundamental linear algebra operations.

Scaling The expression, αT_{ij} , scales the matrix T_{ij} by a scalar α .

Addition Addition of the form $T_{ij} = A_{ij} + B_{ij}$. This applies to scalar, vector, and matrix.

Hadamard Product Hadamard product, also called point-wise product or dot product, of the form $T_{ij} = A_{ij} \circ B_{ij}$. 'o' denotes the point-wise product between two operands.

Permutation/Transposition $T_{ij} = T_{ji}$ is a transpose operation $T_{ij} = (T_{ij})'$ where ' ' denotes the transposition operator.

Single Inner Product The single inner product operator is denoted by ' $\bullet$ '. $s = A_i \bullet B_i$ represents a vector-vector inner product, and is a shorthand notation for $s = \sum_i A_i B_i$. $T_i = A_{ij} \bullet B_j$ represents a matrix-vector multiplication

($T_i = \sum_j A_{ij}B_j$). Matrix-matrix multiplication, $T_{ij} = A_{ik} \bullet B_{kj}$ (i.e. $T_{ij} = \sum_k A_{ik}B_{kj}$), is also recognized.

Double Inner Product $s = A_{ij} \cdot \cdot B_{ij}$ (i.e. $s = \sum_i \sum_j A_{ij}B_{ij}$) is recognized as a double inner product where the operator is denoted by ‘ $\cdot \cdot$ ’.

Kronecker Product $T_{ij} = A_i \otimes B_j$ is recognized as an outer product operation where the Kronecker operator is denoted by the symbol ‘ $\otimes$ ’.

Apart from the above linear algebra operations, the following operations are also supported.

Reduction Reduction patterns such as sum reduction $s = \sum_i A_i$ and product reduction $s = \prod_i A_i$.

Functional Mapping Mapping of the form $T_i = f(A_i)$, $T_{ij} = f(A_{ij})$ and so on.

The function $f(\cdot)$ can be a mathematical operation such as a trigonometric function (sine, cosine, tangent, $\dots$), as well as other operations such as *abs* and *log*.

3.3 Linear Algebra Language

It is difficult for a pattern recognition framework to cover all possible patterns. This often leads the user to modify the original source code to enable the framework to execute correctly. A domain-specific approach can address the goals of hiding the complexity of programming for different hardware, and express the linear algebra operations directly. A domain-specific language is a computer programming language of restricted expressiveness focused on a particular domain. The low-level complexity of parallel programming on the hardware will be hidden by the domain-specific language. Therefore the programmer only needs to write using domain-specific notation and constructs. The domain-specific language codes will appear sequential, and the programmer can still reuse the original software process and flow in the domain-specific language application development.

```

1 def function(in n:Int, in T:Float[n][n], in c:Float[n],
2             inout x:Float[n]){
3     var x_old:Float[n];
4     x_old[0:n:1] = x[0:n:1];
5     x[0:n:1] += T[0:n:1][0:n:1] ** x_old[0:n:1];
6     x[0:n:1] += c[0:n:1];
7 }

```

Figure 3.2: LAL code based on Figure 1.1.

We have developed a Linear Algebra Language (LAL), aiming to solve this problem. It provides a platform-independent way of representing the linear algebra operations using a domain specific language. There are two ways of utilizing LAL for writing linear algebra operations. Firstly, the user can write naïve codes, use the extraction engine to identify the linear algebra operations and transform the operations to LAL. Secondly, the user can also write linear algebra kernels directly using LAL.

Figure 3.2 shows an example of LAL code based on Figure 1.1. It defines four function parameters: n is the input size, T is a matrix of n^2 elements, c and x are vectors of size n . n , T and c are defined as input parameters, while x is defined as an input and output parameter. A temporary vector x_old is defined in the function. All elements in x are copied into x_old . The matrix multiplication of T and x_old is assigned to x . Finally, vector addition of c and x is performed.

3.4 Summary

This chapter gives a brief overview of the code transformation framework in this thesis, with an illustration of the framework architecture. It describes the linear algebra operations supported in the framework. A domain-specific language, the Linear Algebra Language (LAL), is also introduced, which is used to express the linear algebra operations in the abstract syntax tree. The next chapter will give details of the extraction process which identifies the linear algebra operations from the existing source code.

Chapter 4

Extraction Engine

In this chapter, the extraction engine is described in greater detail. The purpose of the extraction engine is to take a loop nest and identify all the parts that are linear algebra operations. It does this by a series of steps that include extracting critical information from the code and performing pattern matching on the expressions to extract the linear algebra operations.

The extraction and recognition process consists of the following steps:

1. Section 4.1 describes how the user can annotate their codes with the relevant information regarding the dimension and size of the vectors and matrices.
2. There are three components that can be used for domain computation and shape recognition of the operands. Section 4.2.1 (**Loop Extraction**) describes the computation of the domain through the loops in the program. Section 4.2.2 (**User Annotated Loops**) describes the use of user annotations, such as OpenMP or OpenACC pragmas, to identify possible loops for extraction. Section 4.2.3 (**Polyhedral Extraction**) describes the domain computation using the polyhedral model instead of the loop structure.
3. Section 4.3 describes how to perform canonical reduction before recognition is carried out.
4. Section 4.4 describes the procedures for recognizing linear algebra operations.

4.1 Array Size Annotation

```
1 #pragma array{A[m][n]}  
2 float* A = malloc(n*m*sizeof(float));
```

(a) The pragma defines an array, A , of dimension 2, and with m rows and n columns.

```
1 #pragma array{A[n], B[m]}  
2 int calculate(int *A, int *B);
```

(b) The pragma defines a column vector A with n elements, and a column vector B with m elements.

Figure 4.1: Array Size Annotation for array declarations.

For C/C++ programs, when dynamic arrays are allocated, only the total size of the array needs to be specified. However, this information is not sufficient to deduce the shape of the original array. For example in Figure 4.1a, when the malloc function is used to declare a dynamic array, only the total size of the array is specified in bytes, but not the shape of the array nor the size of each dimension. In this case, the framework cannot determine the shape of the array; it cannot determine whether n or m is the number of rows or columns from the original source code. Therefore we propose the addition of user annotations, to supplement information that cannot be derived directly from the source code. The user annotations are necessary to define the original size and dimensions of the array.

We have designed a pragma to declare the size and dimension for each array used in the application. Such a pragma can be placed either at memory allocation (Figure 4.1a) or at function declaration (Figure 4.1b). Figure 4.2 describes the context-free Extended BackusNaur Form (EBNF) grammar for writing the array size annotation. By using the directives, hardware-specific details related to memory

```
Array-List ::= Array-Declarator1 { ',' Array-Declarator2 }  
Array-Declarator ::= Identifier Array-Size1 { Array-Size2 }  
Array-Size ::= '[' SimpleExpr ']'
```

Figure 4.2: Context-free grammar for array size annotation.

allocation and data alignment can be hidden from the user. The hardware-specific optimization will be handled by the vendor libraries. This also improves platform portability as the user does not need to explicitly add data alignment to their codes.

4.2 Domain and Shape Recognition for Arrays

4.2.1 Loop Extraction

It is crucial to first identify domain related information for each statement in a loop nest. The loop extraction occurs in a series of steps:

1. Loop normalization
2. Loop fission
3. Shape Recognition

An example illustrating the stages in the loop extraction is shown in Figure 4.3.

First, loop normalization is carried out to transform all the loops (*do loops*, *while loops* and *for loops*) to a canonical form. After transformation of the loops, the iteration domains are normalized, which shifts the starting index to zero and the increment to one. This ensures that the iteration domains of each statement in the loop body can be identified and merged easily. In Figure 4.3 shows the loop extraction steps on the original code in Figure 4.3a. The normalized code is shown in Figure 4.3b.

Next, the loop extraction component performs loop fission, also known as loop distribution [22], as shown in Figure 4.3c before extracting the linear algebra operations in Figure 4.3d. The goal of the loop fission is mainly to separate the statements in the loops into individual loops; transforming the loop-carried dependencies into loop-independent dependencies. Statements with loop-carried dependencies have dependencies between the statements in the loop, but in different loop iterations. By transforming to loop-independent dependencies, the statements in each iteration

```

1  for (int i = k; i < n; i++){
2    for (int j = k; j < n; j++){
3      for (int m = k; m < n; m++)
4        A[i][j] += Q[i-k][m-k]*T[m-k][j-k];
5    }
6    hv[i-k] = b[i];
7  }

```

(a) Original code fragment.

```

1  for (int i = 0; i < n-k; i++){
2    for (int j = 0; j < n-k; j++){
3      for (int m = 0; m < n-k; m++)
4        A[i+k][j+k] += Q[i][m]*T[m][j];
5    }
6    hv[i] = b[i+k];
7  }

```

(b) After loop normalization.

```

1  for (int i = 0; i < n-k; i++)
2    for (int j = 0; j < n-k; j++)
3      for (int m = 0; m < n-k; m++)
4        A[i+k][j+k] += Q[i][m]*T[m][j];
5  for (int i = 0; i < n-k; i++)
6    hv[i] = b[i+k];

```

(c) Loop fission on statements $\mathcal{S}_4$ and $\mathcal{S}_6$.

$$A[k, n; k, n] = Q[0, n - k; 0, n - k] \bullet T[0, n - k; 0, n - k]$$

$$hv[0, n - k] = b[k, n]$$

(d) Extracted linear algebra operations.

Figure 4.3: An example illustrating the stages in the *Loop Extraction*.

```

1  for(int i = 0; i < N; i++){
2    a[i] = b[i];
3    c[i] = a[i-1];
4  }

```

(a) Dependence from line 2 to 3 due to array a crosses iterations of the i loop.

```

1  for(int i = 0; i < N; i++){
2    a[i] = b[i];
3  }
4  for(int i = 0; i < N; i++){
5    c[i] = a[i-1];
6  }

```

(b) After loop fission, the dependence is no longer carried by any loop.

Figure 4.4: Example of using loop fission to remove loop-carried dependence.

can be executed independently. An example of the loop fission transformation is shown in Figure 4.4.

Before performing loop fission, data dependency analysis is carried out in order to determine if different loop iterations are independent. Let $\mathcal{S}_k$ be a particular statement within a loop nest and let $\mathbf{v}$ be the iteration vector of the loop nest enclosing the statement $\mathcal{S}_k$ from the outermost to innermost loop. For example, the iteration vector of the loop nest with i and j loops is $\mathbf{v} = (i, j)$. Each array reference that is read or written has an assigned iteration vector $\mathbf{v}$ whose length depends on the dimension of the array. The iteration domains $\mathcal{D}^{\mathcal{S}_k}$ of each statement $\mathcal{S}_k$, $\mathcal{D}^{\mathcal{S}_k} = \{\mathbf{v} \mid L_i \leq v_i < U_i, \forall i = 1 \dots l\}$, where L_i and U_i denote the lower and upper bound of each loop and l is the dimensionality of $\mathbf{v}$, are also identified. Dependencies are obtained through static analysis by using an external library [23].

After data dependency has been carried out, the loop fission stage attempts to break up the loops into multiple loops with the same iteration domain. If there are no flow dependencies and loop-carried dependencies between the statements within

the loop nest, the loop can be safely split into multiple loops. Statements with dependencies are placed within the same loop nest, while the statements with no dependencies are split into separate loops. By splitting the loops, it is easier to perform pattern matching on each statement. In Figure 4.3b, the statements $\mathcal{S}_4$ and $\mathcal{S}_6$ have no data dependencies. Therefore, they can be split into two loop nests, the first having the iteration domain $\mathcal{D}^{\mathcal{S}_4} = \{(i, j, m) \mid 0 \leq i, j, m < n - k\}$, and the second statement having the iteration domain $\mathcal{D}^{\mathcal{S}_6} = \{i \mid 0 \leq i < n - k\}$.

After loop fission, linear algebra operations can be extracted from the statements in each parallel loop. For those statements that can be extracted to linear algebra operations, their loops are extracted to a single statement involving the linear algebra operations.

Currently, the extraction is limited only to static control programs [24]. Static control programs are a sub-class of imperative programs that includes imperfectly nested loops. They must satisfy the following constraints:

- The loop counters are integers, and they are incremented by constant steps.
- The loops are bounded, and each bound must be an affine expression of the value of the outer loop counters, the parameters and the constant values, e.g. array sizes.
- The statements enclosed within the loops access multi-dimensional arrays (this includes scalars and one-dimensional arrays) through multi-dimensional affine of the loop counters and the parameters.

An Affine expression refers to the sum of a linear expression and a constant. For example $a * x + b$, where a and b are constants, x is a variable and $a * x$ is the linear expression. A multi-dimensional affine expression consists of a list of affine expressions.

The iteration domains of the loops are extracted and are used to identify the shape of the arrays. According to the dimension of the iteration domain, $\mathcal{D}^{\mathcal{S}_k}$, we can deduce the dimensionality of the array. This will determine whether the array is

```

1 for (int i = 0; i < N; i++) {
2   for (int j = 0; j < N; j++) {
3     r[i][j] = p[i] * q[j];
4   }
5 }

```

Figure 4.5: Shape recognition to identify N^2 elements in r array and N elements in p and q arrays.

a scalar, vector or matrix. The bounds of the iteration domain are used to identify the size of the array expressions in the statement.

For example in Figure 4.5, the iteration domain of statement $\mathcal{S}_3$ is $\mathcal{D}^{\mathcal{S}_3} = \{(i, j) \mid 0 \leq i, j < N\}$, where $\mathbf{v} = (i, j)$. Since the dimension of $\mathbf{v}$ is 2, the r array is identified as a matrix, while p and q are identified as vectors. According to the iteration domain, r has a sub-matrix of N^2 elements, while both p and q contain at least N elements. The exact size of the arrays are defined by the user annotation. The shape recognition only identifies the sub-array involved in the statement, which can be a slice of the original array defined.

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.

4.2.2 User Annotated Loops

Other than automatically extracting linear algebra operations from the loops, it is also possible for the user to annotate the regions in their codes for extraction. The directive-based approach taken by both OpenMP and OpenACC provides a simple but powerful way for users to quickly parallelize their code. 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 multi-core 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

```

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

```

Figure 4.6: OpenACC code with two dimensional parallel loops.

containing nested loops are usually good candidates for parallelization on multi-core processors or GPUs, and are annotated using `#pragma omp parallel for` and `#pragma acc kernels loop` for OpenMP and OpenACC, respectively. Figure 4.6 shows the OpenACC annotation for a two-dimensional parallel loop.

Since the users have annotated the loops in their application, these loops provide opportunities for extraction. Given that the users have already explicitly declared the loop to be parallel, there is no need to perform any loop dependency checks on the loop. However, the user is still required to annotate the arrays with the relevant size and dimension as described in Section 4.1. The iteration domain of each statement is extracted in a similar way to that in Section 4.2.1 to identify the shape of the sub-arrays in the statement.

We also extract information regarding the data transfer between the host and the GPU from the OpenACC data copy directives. 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.

4.2.3 Polyhedral-based Extraction

Classical methods of loop transformation are not scalable compared to the recent work on polyhedral optimizations [25, 26]. The approach described in Section 4.2.1 can currently deal only with loops with affine expressions which is yet another well-known problem that polyhedral optimizations attempt to solve. It is possible to unify the loop transformations and extraction of the domain information using the polyhedral model.

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 by 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

$$D^S = \{\mathbf{i} | \exists \mathbf{i}_{lv}, \Lambda^S \times [\mathbf{i}, \mathbf{i}_{lv}, \mathbf{i}_{gp}, 1]' \geq 0\}. \quad (4.1)$$

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_1} is defined such that an instance of S_1 given by a particular $\mathbf{i}_{k_1}$ is executed before another instance $\mathbf{i}_{k_2}$ of statement S_2 if and only if

$$\Theta^{S_1} \times [\mathbf{i}_{k_1}, \mathbf{i}_{lv}, \mathbf{i}_{gp}, 1]' \ll \Theta^{S_2} \times [\mathbf{i}_{k_2}, \mathbf{i}_{lv}, \mathbf{i}_{gp}, 1]' \quad (4.2)$$

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

$$f(\mathbf{i}) = F \times [\mathbf{i}, \mathbf{i}_{lv}, \mathbf{i}_{gp}, 1]'. \quad (4.3)$$

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 4.1. 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 sub-array or a slice of the original array A .

For example, the iteration domain of a statement S is shown in Equation 4.4. An array A with the expression $A[i-1]$ has the access function $\begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$. Applying the access function to Λ^S gives the accessed domain D^A in Equation 4.5. Since the linear constraints only contain i , A is identified as a vector with the domain: $0 \leq i \leq N - 2$.

$$\Lambda^S = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \end{bmatrix} \begin{matrix} 0 \leq i \\ i \leq N - 1 \end{matrix} \quad (4.4)$$

$$D^A = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -2 \end{bmatrix} \begin{matrix} 0 \leq i \\ i \leq N-2 \end{matrix} \quad (4.5)$$

Similar to Section 4.2.1, only general square matrices are recognized.

4.3 Canonical Reduction

Before performing algorithmic recognition in the next step, 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 shown below 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 associated iteration domain information that is related to its enclosing statement and determined previously. 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.

The order of operation defines the order in which the expression will be evaluated. The precedence level of operators is as follows:

Reduction Sum ($\sum$) and Product ($\prod$)

Unary Transposition ($'$)

Matrix Dot Product ($\circ$), Outer Product ($\otimes$), Matrix Product ($\bullet$)

Binary Times ($*$), Plus ($+$)

4.4 Linear Algebra Recognition

In this step, the linear algebra operations are recognized and converted into algebra form. Extracting the linear algebra operations from the statement relies on matching the expression tree of a single statement with a series of rules. Matching is performed

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 algebra form after extending with outer product:

$$A \leftarrow x \otimes 1_n$$

Figure 4.7: 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 algebra form after applying permutation to the indices:

$$A_{i,j} \not\leftarrow B_{j,i} \Rightarrow A_{i,j} \leftarrow (B^T)_{i,j}$$

Figure 4.8: Code fragment that permutes the indices to match the domains.

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 D^S has already been determined previously.

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

1. If the dimensions of the variables on each side of the assignment do 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 do not match, the vector of loop indices, $\mathbf{i}$, 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 4.7. Figure 4.8 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_{lhs}^S and D_{rhs}^S . 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 expression is multiplication, the expression returned is determined by the dimension of D_{lhs}^S , as shown in Table 4.1.

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

Table 4.1: Dimension of D_{lhs}^S determines the operator under the Reduction rule.

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 iterations 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.

4.5 Summary

The extraction method is not restricted to any programming model as long as the language supports arrays and loops. After linear algebra operations have been extracted, it can be expressed in terms of LAL. The linear algebra operations are then fed to the transformation process for code transformation as described in the next chapter.

Chapter 5

Transformation Engine

The *Transformation Engine* which transforms the linear algebra operations to respective library calls is explained in detail in this chapter.

There are many available linear algebra libraries that allow the user to write in a more natural way, i.e. write codes in terms of linear algebra expressions. These libraries make use of C++ template meta-programming to provide matrix or vector classes with overloaded linear algebra operators. Code in this form is also closely similar to the linear algebra intermediate representation. To avoid reinventing the wheel, we make use of existing libraries, and develop a transformation component targeting these libraries. Section 5.1 elaborates on this component in detail.

Although C++ template libraries are more readable to the user, there are fundamental drawbacks to the use of templates. In Section 5.2, we extend the transformation engine to directly generate function calls to vendor-optimized linear algebra libraries for different platforms. This will allow the user to utilize vendor-optimized libraries directly without using the linear algebra template libraries. The transformation engine will be modular, allowing the addition of further modules to cater for multiple platforms.

5.1 Linear Algebra Template Libraries

The first component supports code generation to three different platforms: (i) the vector processing unit, (ii) the multi-core processor and (iii) the GPU. We utilize different libraries for each target, and transform the linear algebra operations into codes using the programming interfaces in these libraries.

5.1.1 Vector Processing Unit

The code generation for vector processing units is implemented using the Eigen C++ Template library [4]. Eigen generates explicit vectorization for a variety of vector processing units, including Intel SSE, and falls back to non-vectorized code if there are no equivalent instructions.

One- and two-dimensional arrays are mapped directly to the Eigen vector and matrix objects. The vector objects are matrix objects with only one column. Fixed-size Eigen objects enable loop unrolling if the size is small. Most operations are able to be mapped to the overloaded operators or member function calls in Eigen. For example, a single inner-product or matrix multiplication in Eigen uses the `*` arithmetic operator, e.g. `A = B * C`, while the transposition operator is mapped to the member function `B.transpose()`. The detailed table of the operator mapping is shown in Table 5.1

Linear Algebra Operator	Eigen Operator/Function Call
Sum	<code>sum()</code>
Product	<code>prod()</code>
Transposition	<code>transpose()</code>
Dot Product	<code>dot()</code>
Outer Product	<code>x * transpose(y)</code>
Matrix Product	<code>*</code>
Times	<code>*</code>
Plus	<code>+</code>

Table 5.1: Linear Algebra Operators mapping for Eigen

5.1.2 Multi-core Processor

The code generation for multi-core processors is implemented using the Armadillo C++ linear algebra library [21]. Internally, Armadillo does not support multi-core processors. However, by using multi-threaded vendor-tuned implementations of BLAS, Armadillo is able to make full use of the multi-core processors for computation. The main performance of Armadillo depends on the underlying implementation of BLAS to which it is linked, such as the Intel MKL.

Armadillo supports arrays up to three dimensions, which are mapped to the vector, matrix and cube objects. The list of the operators mapped is shown in Table 5.2. If there are multiple matrix multiplications in an expression, Armadillo applies matrix chain optimization to select the most efficient way to multiply these matrices together.

Linear Algebra Operator	Armadillo Operator/Function Call
Sum	sum()
Product	prod()
Transposition	trans()
Dot Product	dot()
Outer Product	$x * \text{trans}(y)$
Matrix Product	*
Times	%
Plus	+

Table 5.2: Linear Algebra Operators mapping for Armadillo

5.1.3 Graphics Processing Unit

For the GPU target, we did not find a C++ template library that meets our requirements. Therefore we modified Armadillo: replacing the original BLAS calls to the corresponding CUBLAS [5] calls. GPU initialization that is required prior to invocation of the CUBLAS routines, as well as clean-up at the end of the application life-cycle, are added to the start-up function. For each vector or matrix variable, a new device memory pointer of the appropriate type is instantiated, followed by a call to allocate space on the device. The linear algebra operations which are identified

Linear Algebra Operator	CUBLAS routines
Sum	Not supported
Product	Not supported
Transposition	Part of function calls
Dot Product	dot()
Outer Product	ger()
Matrix Product	gemv()/gemm()
Times	Not supported
Plus	Not supported

Table 5.3: Linear Algebra Operators mapping for CUBLAS routines

by the extraction layer are decomposed into individual operations and replaced by corresponding calls to a CUBLAS routine. The list of BLAS routines that map to the linear algebra operators is shown in Table 5.3. For those operators that are not mapped to BLAS routines, we define custom GPU kernels for the operators. Prior to making a CUBLAS call, the variables involved in the CUBLAS routine need to be copied from the CPU memory to the device memory if the variable is not available in the GPU memory through a previous copy. Once the result is evaluated, the resultant vector or matrix is copied from the GPU memory back to the CPU memory. Currently, we do not minimize data transfers across statements.

5.1.4 Adaptive Mapping

Given that the framework is able to generate codes for three different platforms, there is the possibility of automatically selecting the appropriate platform according to the problem size. The rationale for adaptive mapping is that if we are able to switch the execution of each linear algebra operation to different hardware according to the problem size, we may be able to harness the higher performance benefit of the different libraries on each hardware platform.

A major benefit of this framework is that, because of the consistency of using a single representation, the linear algebra form, we can target each individual operation on different platforms. In general, although this may not give the best performance compared to hand-tuned code, the framework greatly simplifies the process of targeting code for different platforms, as well as reduces the cost of maintaining

them.

By adaptive mapping, it means that all the necessary code paths to different hardware targets are generated together and included simultaneously in the execution binary. Only the branching or switching points are decided statically before run time. In this manner, because all the code paths are included in the execution binary, the hardware target to use for each operation can be decided on-the-fly during run time. To be able to do this, the execution cost of each operation on each platform must be known *a priori*. Note that adaptive mapping is useful because the size of each operation may not be known at compile-time, and so the optimal hardware platform cannot be chosen at compile-time.

Our framework is extended to support adaptive mapping by taking the linear algebra operations and deciding on the target platform. In order to carry out the mapping, a cost model for translating the linear algebra operations is necessary. We use an empirical cost model instead of an analytical one for the framework. An analytical model is not the preferred approach owing to the complexity of modelling the effects of modern hardware which has features such as speculative execution, out-of-order execution, and branch prediction. Furthermore, as discussed in the previous section, each code generator may contain different code optimization paths for its targeted hardware. All of these make designing a single analytical model challenging. We have taken the empirical approach by treating each code generator as a black box. The execution cost of each mathematical operation, parametrized by the data type, the problem size, and the target hardware, is stored in a database. These costs are derived by running micro-benchmarks on each of the target platforms.

Let the cost of an operation, op , on a particular target, $target$, be denoted by $C_{target,op,type}(N)$ where N is the problem size, $target \in \{vector, multicore, gpu\}$, $type \in \{float, double\}$, and op represents the mathematical operation. When not referring to a specific architecture or data type, the cost is denoted by $C_{op}(N)$. As an example, $C_{vector,dotprod,float}(100)$ denotes the execution cost of a floating-point dot-product with a problem size of 100 on a vector processing unit.

Figure 5.1: A piecewise linear cost model.

The execution cost $C_{op}(N)$ for a particular operation varies with the problem size. We represent this by a piecewise linear model

$$C_{op}(N) = a_i \times N + b_i, \text{ for } n_i \leq N < n_{i+1},$$

where $[n_i, n_{i+1})$ represents the range of the i th segment, and a_i and b_i are the gradient and intercept of the segment. The model is illustrated in Figure 5.1. Internally, the piecewise linear model is stored in the database as a sequence of 3-tuples (n_i, a_i, b_i) .

For a compound expression with multiple operations, the mapping component evaluates the total execution cost of all the operations in order to decide which hardware to target for launching the computations. If the size of the problem is known at compilation time, the component will generate code for exactly one target, i.e. the one with the lowest execution cost. However, if the problem size is only determined at run time, then the generated code must contain different code paths which are optimized for different problem sizes.

5.2 Direct to BLAS

There are disadvantages of using C++ template libraries. Since the compiler generates additional code for each template type, the template libraries introduce ad-

ditional overhead due to the increase in code size. It can also be difficult to debug code that is developed using templates. Since the compiler replaces the templates, it becomes difficult for the debugger to locate the code at runtime. Performance is dependent on the implementation of the template libraries that is linked to the vendor-optimized BLAS library. In cases of using BLAS library for different platforms, such as the CUBLAS for Nvidia GPU, many linear algebra template libraries do not have directly support for such libraries.

In this component, the framework will transform the code from the linear algebra intermediate representation to equivalent BLAS function calls. Note that the class of linear algebraic operations is a superset over BLAS operations. However, 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 5.4 shows a list of the BLAS operations that are mapped by our framework. Additional mappings are supplemented in the form of BLAS-like extensions and are listed in Table 5.5. A , B and C denote two-dimensional matrices while x , y and z denote one dimensional vectors. The *Transformation Engine* performs the actual code transformation using the following steps: (a) statement fusion (b) operation mapping, and (c) platform-specific code generation.

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 5.4: BLAS Operations

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

Table 5.5: BLAS-like Extensions

5.2.1 Statement Fusion

Because many of the linear forms listed in Tables 5.4 and 5.5 are made up of compound operations, for 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 [27]. 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 5.2 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 5.1b (0_n denotes the zero vector) is fused with the x on the right-hand-side of line 5.1c, 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 5.2b and 5.2c to remove the intermediate x term which is only used once in the code fragment, resulting in the final form in line 5.3b.

Original statements:

$$x_{old} \leftarrow x \quad (5.1a)$$

$$x \leftarrow 0_n \quad (5.1b)$$

$$x \leftarrow T * x_{old} + x \quad (5.1c)$$

$$x \leftarrow c + x \quad (5.1d)$$

After constant propagation:

$$x_{old} \leftarrow x \quad (5.2a)$$

$$x \leftarrow T * x_{old} \quad (5.2b)$$

$$x \leftarrow c + x \quad (5.2c)$$

Final fused form:

$$x_{old} \leftarrow x \quad (5.3a)$$

$$x \leftarrow T * x_{old} + c \quad (5.3b)$$

Figure 5.2: An illustration of statement fusion.

5.2.2 Operation Mapping

After fusion, the next step performs mapping of the linear algebra forms to equivalent BLAS or BLAS-like routines. This is done by stepping through a series of predefined rewrite rules to generate the final mapping. There are two levels of rewrite rules: (a) statement rules 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 5.6 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.

Order	Match Operator/Pattern	Rule
1	Σ	Sum
2	$\bullet$	Dot Product
3	$\times$	Matrix Multiply
4	$\otimes$	Outer Product
5	$\alpha a + \beta b$	Scaled Addition

Table 5.6: List of patterns and corresponding operation rules.

Since a statement may contain a number of linear algebra operations, it is nec-

essary 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 Table 5.6. 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:

1) Sum Rule

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

2) Dot Product Rule

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

3) 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-matrix operation depending

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 5.7: Different forms of **GEMV** and **GEMM** depending on the dimensions of A and B .

on their dimensionality determined by the Extraction Engine. The equivalent BLAS operations **GEMV** and **GEMM** accept only two operands. This rule is shown in Table 5.7.

4) 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 5.8.

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 5.8: Different forms of **GER** depending on the value of β .

5) 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 (see below) 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

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 5.9: Pattern matching with the Scaled Addition Rule.

constant addition to a matrix (**MATADDCONST**). Otherwise, the expression is mapped to a matrix accumulation operation (**MATACCSCAL**). A summary of the patterns is shown in Table 5.9.

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 5.10.

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

Table 5.10: Different scaled vector operations depending on the value of β .

5.2.3 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. In the following, we provide details about the mapping of BLAS-like extensions to two specific vendor-provided libraries, Intel MKL [3] and Nvidia CUBLAS [5], as they are widely used.

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

Table 5.11: Mapping of BLAS-like extensions to MKL and CUBLAS routines

Intel MKL

Apart from the standard BLAS functions, Intel MKL provides several routines to extend the functionality of its library. These include 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 is shown in Table 5.11. 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.

Nvidia CUBLAS

CUBLAS provides BLAS-like extensions through the following operations shown in Table 5.11. The `geam` 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 5.3a, there is no library call that supports the power function `pow`. Therefore, a sub-kernel as shown in Figure 5.3b is generated. Essentially, sub-

```

1  for(i = 0; i < N; i++)
2  {
3      C[i] = pow(A[i], b);
4  }

```

(a) Original code

```

1  __device__ void VecAdd1D(float* A,
2      const float b, float* C)
3  {
4      int i = threadIdx.x;
5      C[i] = pow(A[i], b);
6  }
7
8  // Kernel call
9  VecAdd1D<<1,N>>(A, b, C);

```

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

Figure 5.3: Generation of non-standard function into CUDA sub-kernels.

kernels are defined as CUDA device-only functions and they are generated at the granularity of a basic block.

For each sub-kernel, the set of variables for each statement in the sub-kernel needs to be computed. Live-variable analysis is performed on the statements [28]. An array is an *in-array* of a given statement if it is live for the statement and is read in the statement. Similarly, an array is an *out-array* if it is live and written in the statement. A scalar variable is defined as *out-variable* if the scalar variable is modified by the statement. The data transfers are minimized by only transferring necessary in-array, out-array and out-variables. A scalar *in-variable* is passed into the sub-kernel as a parameter, with no additional memory transfer between the CPU and GPU needed. For the *in-array* and *out-array* that is read, data transfers from CPU memory to GPU memory for the arrays are inserted at the beginning of the

sub-kernel. For the *out-variable* and *out-array*, the transfers from GPU memory to CPU memory are inserted at all exit points of the kernel. For arrays that are defined and used only in the kernel, they are only allocated on the GPU. For the remaining scalar variables, they are allocated on the host.

5.3 Summary

In this chapter, we describe the code transformation for multiple platforms. The first Section 5.1 shows the code generation to C++ template libraries. Eigen is generated for running on the vector processing unit. For multi-core processors, Armadillo is generated and linked to the multi-threaded Intel MKL. As for Nvidia GPUs, a customized Armadillo is linked to Nvidia CUBLAS. Given three different platforms for the user to execute their application, we present an adaptive mapping to automatically decide which platform to execute the linear algebra operations.

We also extend the framework to generate code directly to equivalent BLAS or BLAS-like operations. Due to the difference in the vendor-provided libraries, we also generate our internal implementation for any missing library calls. Therefore this component is able to generate code for either the Intel MKL or the Nvidia CUBLAS.

In the next chapter, we evaluate the performance of the two components of our code transformation framework.

Chapter 6

Evaluation and Results

In this chapter, we evaluate the effectiveness of our extraction and code transformation framework. The experiment is separated into two parts according to the two different code transformation components. Section 6.1 evaluates the performance of the code generation for linear algebra template libraries. Section 6.1.3 illustrates the use of adaptive mapping between different platforms and presents results for it. In Section 6.2, the code transformation component, Direct to BLAS (DTB), that generates BLAS function calls directly is evaluated.

The following codes are used for benchmarks:

cg Conjugate Gradient method, without preconditioning, that solves a symmetric and positive-definite system of linear equations.

gmres Generalized Minimal Residual method, without preconditioning, that uses an iterative method to solve a non-symmetric system of linear equations.

jacobi A classical 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 for solving linear least squares problems.

These are general codes are not specially tuned for any particular hardware. They are taken from the following online sources:

- *cg* and *gmres* are taken from the Iterative Methods Library (IML++, <http://math.nist.gov/impl++/>),
- *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/>).

6.1 Linear Algebra Template Libraries

6.1.1 Setup

We perform experiments on a workstation equipped with an Intel Xeon E5540 processor, which support SSE4.2 extensions. The workstation is also equipped with a Nvidia Tesla GeForce 680 graphics card. Table 6.1 shows the specification of the hardware used in our experiments.

	CPU	GPU
Architecture	Intel Xeon E5540	Nvidia GeForce GTX 680
Number of cores	4	1536
Clock rate	2.53GHz	1006MHz
Size of memory	12GB	2GB
Memory bandwidth	8GB/s	192.3GB/s

Table 6.1: Hardware Configuration.

Figure 6.1: Experimental results for *cg*, *gmres* and *jacobi*.

Figure 6.2: Experimental results for *matexp*, *mcl* and *qr*.

Benchmarks	Loops	Non-abstractable Loops	Abstracted Statements
<i>cg</i>	14	1	13
<i>gmres</i>	28	7	19
<i>jacobi</i>	3	0	4
<i>matxp</i>	18	4	9
<i>mcl</i>	14	3	8
<i>qr</i>	17	1	10

Table 6.2: Summary of the number of loops in each benchmark, the number of loops that cannot be abstracted, and the number of statements that are abstracted into linear algebra form.

6.1.2 Evaluation

To evaluate the framework, the *Loop Extraction* component is applied to each of the benchmarks, and generates codes for each of the three supported targets. Table 6.2 summarizes the number of high-level abstractions that are performed; it shows the total number of loops in each of the original benchmarks, the number of loops that cannot be abstracted, and also the number of statements that are identified and lifted up to linear algebra form. For instance, the *cg* benchmark has a total of 14 loops, of which 13 of the loops are lifted, while only one loop cannot be abstracted. Note that the last two columns of the table do not necessarily add up to the second column because of the loop fission and abstraction stages.

For each of the benchmarks and for a given problem size, we obtain measurements for each of the three targets and compare them with the original sequential version to obtain the speedup (or slowdown) factor. The first generated target exclusively targets the vector processing units using the Eigen library. The second generated target is executed on multi-core processors using Armadillo linked to the Intel MKL library. The third generated target is executed on the GPU using the modified Armadillo library linked to CUBLAS.

In Figure 6.1 and 6.2, we plot the speedup (> 1) or slowdown (< 1) achieved by each of the code generators over the original baseline implementation. The horizontal axis denotes the variation in problem sizes. For *cg*, *gmres*, and *jacobi*, problem size refers to the number of linear equations to be solved. For *matexp*, *mcl*, and

qr , it refers to the size of the matrix. In the figure, the results for different code generators are labelled ‘Eigen’, ‘Armadillo-MKL’, ‘Armadillo-CUBLAS’ to denote vector processing unit-only, multi-core processor-only, and GPU-only execution respectively.

Figure 6.1 and 6.2 demonstrate that the code generated for different targets do not perform uniformly across different problem sizes, and that different platforms are suitable for different problem scales. In most cases (except for *matexp*), it can be observed that Eigen code is faster for small problem sizes. On the other hand, depending on the application, Armadillo-MKL or Armadillo-CUBLAS is faster when the problem size is larger.

One of the main reason why Eigen is faster for small sizes is that it does not have a startup cost compared to Intel MKL or CUBLAS. The Intel MKL library has an associated thread startup cost while the CUBLAS library incurs a penalty during kernel launch. Due to the high overhead of data transfers, the CUBLAS underperforms both the Eigen and Armadillo-MKL for three benchmarks, *cg*, *gmres* and *jacobi*. CUBLAS typically performs best for large problem sizes, and when there are sufficient computations to mitigate the cost of transferring data from the host to the GPU.

Results for *matexp*, *mcl* and *qr* show that for large problem sizes of 500 and above (800 for *qr*), where the number of computations are significant, the GPU target can achieve speedups that are greater than the Eigen and Armadillo-MKL targets. The number of computations for these benchmarks are significant to due the large number of matrix-matrix multiplication operations that are required by the algorithm.

6.1.3 Adaptive Mapping

From the experimental results in the Section 6.1.2, there is no single hardware platform that has the best performance across different problem sizes. The framework is shown to simplify the process of re-targeting user code on multiple platforms. In

Figure 6.3: Achieving adaptive mapping for expressions with multiple operations.

this section, we will study and experiment with an adaptive technique for mapping the linear algebra operation to different hardware platforms.

To illustrate the construction of the database for adaptive mapping, Figure 6.3 plots the total execution cost of an expression with two operations, matrix addition (+) and matrix multiplication ($\bullet$), i.e. $C_{tot}(N) = C_+(N) + C_\bullet(N)$, for all three targets – vector processing unit, multi-core processor and GPU. These targets employ the Eigen, Armadillo (linked with Intel MKL) and Armadillo (linked with CUBLAS) respectively. Note that the total cost for each target is derived from the micro-benchmarks obtained on the hardware configuration in Table 6.1. As the total execution cost varies when the size of matrix increases, the models intersect at $N = N_1$ and $N = N_2$. Therefore, the transformation engine will generate code that runs on the vector processing unit when N is less than N_1 , and code that runs on the multi-core processor when N is between N_1 and N_2 . Code is generated to run on the GPU when N is greater than N_2 . In this way, the code generated will contain different execution paths depending on the problem size, and the optimum target will be selected for a given input size. Only one version of the code is generated if the problem size is known.

Size	Fastest Library		Adaptive Mapping
	Library	Speedup	Speedup
50	Eigen	3.57	3.04
100	Armadillo-MKL	8.36	8.65
200	Armadillo-MKL	16.02	16.18
500	Armadillo-MKL	25.17	33.31
800	Armadillo-CUBLAS	96.35	138.42
1000	Armadillo-CUBLAS	315.87	430.41

Table 6.3: Comparison of speedup between the fastest libraries for each size and adaptive mapping for the *qr* benchmark. Baseline is the original sequential code.

Results

In this subsection, we present the results of using adaptive mapping on the *qr* benchmark. The adaptive code is executed on the same hardware specified in the Table 6.1. In Table 6.3, we tabulate the speedup achieved for different problem sizes. For each size, the best speedup achieved by a particular library (Eigen, Armadillo-MKL, and Armadillo-CUBLAS) is recorded (see “*Fastest Library*” column). The “*Adaptive Mapping*” column lists the speedup achieved if adaptive mapping is used.

The table shows that the adaptive mapping version performs better than the fastest libraries at most sizes, except for the problem size of 50. This implies that the run-time adaptive mapping used in our framework can adapt efficiently to varying loads by launching the computations on targets which have the least execution cost. To examine the contribution to the speedup for the adaptive mapping, we look at the breakdown of each code path in the adaptive mapping in Figure 6.4.

For each problem size, there are two bars plotted. The left bar indicates the run-time when a single library is used (single code path). The right bar indicates the run-time of the adaptive mapping (multiple code paths), with a breakdown for each code path. In Figure 6.4, the run-times are normalized by the right bar. For sizes 50, the left bar represents Eigen execution. For sizes 100, 200 and 500, the left bars represent Armadillo-MKL execution, while for sizes 800 and 1000, the left bars show Armadillo-CUBLAS execution.

For a size of 100 with adaptive mapping, 82.1% of the run-time is executed using

Figure 6.4: Breakdown of run-time for each code path. The right bars show the breakdown for the adaptive mapping. The left bars (dashed outline) show the increase in runtime if only a single code path is taken.

the Armadillo-MKL, while the remaining is executed with the Eigen code path. We can see that the total run-time increases by 13.1% if only Armadillo-MKL is used. Similarly, at a size of 1000, we can see that the run-time is increased by 35% when only Armadillo-CUBLAS is used (left bar), in contrast with the adaptive mapping result (right bar). This is because the adaptive code makes use of Eigen and Armadillo-MKL when it is more efficient to do so. These experiments show that through adaptive mapping, we can achieve the better performance in most cases by selecting the library that performs the best at run-time.

Appendix B compares the results of the adaptive mapping and other code generators for the benchmarks shown in Figure 6.1 and 6.2.

6.2 Direct to BLAS

In our experiments, two possible platforms are supported by the code generation directly to BLAS function calls: Intel MKL for multi-core processors and Nvidia CUBLAS for GPUs. Codes are generated to the corresponding interfaces provided by these two libraries. We refer to codes generated by the framework as *Direct to BLAS* (DTB). DTB-MKL and DTB-CUBLAS are used to denote codes generated for Intel MKL [3] and Nvidia CUBLAS [5] by the framework respectively.

6.2.1 Setup

We conduct 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

	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

Table 6.4: Hardware configuration.

Table 6.4. The number of threads assigned for the MKL experiments is 16 and corresponds to the number of physical cores available.

For these benchmark codes, we add simple compiler directives to the parallelizable loops, targeting multi-core CPUs with OpenMP and GPUs with OpenACC. Table 6.5 shows for each of the benchmarks, the size of input matrices, the number of loops in the code, the number of loops that can be parallelized using compiler directives, and the number of library calls that are generated by the framework.

To evaluate our framework, experiments are conducted to compare performances on both the CPU and GPU. On each hardware target, we compare the performance between simple compiler directive annotated codes and our framework generated codes (DTB). 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 (DTB-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.

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

Table 6.5: Summary of parallelizable loops in benchmarks that are annotated and number of library calls generated.

Benchmark	Original	OpenMP		DTB-MKL	
	Time	Time	Speedup	Time	Speedup
<i>cg</i>	31,659	24,191	1.31	21,964	1.44
<i>gmres</i>	30,566	33,325	0.92	27,292	1.12
<i>jacobi</i>	2,546	1,528	1.67	1,463	1.74
<i>matexp</i>	876	1,207	0.73	336	2.61
<i>mcl</i>	12,055	17,732	0.68	1,812	6.65
<i>qr</i>	3,423	792	4.32	123	27.75

Benchmark	Original	OpenACC		DTB-CUBLAS	
	Time	Time	Speedup	Time	Speedup
<i>cg</i>	31,659	8,019	3.95	8,021	3.95
<i>gmres</i>	30,566	19,435	1.57	15,974	1.91
<i>jacobi</i>	2,546	673	3.78	669	3.80
<i>matexp</i>	876	1,073	0.82	59	14.63
<i>mcl</i>	12,055	18,883	0.64	10,628	1.13
<i>qr</i>	3,423	243	14.07	90	38.03

Table 6.6: 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 DTB-MKL compiled codes. On the GPU, comparisons are made between OpenACC directive and DTB-CUBLAS compiled codes. Run times are recorded in milliseconds.

6.2.2 Performance Comparison

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

CPU Performance

As mentioned previously, OpenMP directives are used for the CPU benchmark. We annotate parallelizable loops with `omp parallel for` directives, with an additional `collapse` clause for two-dimensional loops. The reduction loops are also annotated with a `reduction` directive. The OpenMP annotated codes are compiled using Intel C++ Composer XE 2013 service pack 1 (ICC). As a baseline reference for comparison, we compile 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 6.6, we compare the relative performance of the framework generated code (denoted as

DTB-MKL), and the OpenMP compiler generated code (denoted as OpenMP) to the original ICC compiled un-annotated code.

For many of the benchmarks, ICC manages 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 is because of additional statements in the loop nest which the compiler fails to abstract. For the *qr* benchmark, the operations involve sub-matrices which the compiler is not able to detect. When OpenMP directives are added to the benchmarks, the compiler generates threaded code directly from the loops nests. In general, the threaded versions run faster than the non-threaded versions. This can be seen from the speedups achieved in Table 6.6. However, there are also cases such as *matexp* and *mcl* where thread overheads and poor tiling and code generation from the OpenMP compiler result in worse performance.

In all benchmarks, the DTB-MKL results show better performance than the OpenMP compiled results. The main reason is because the BLAS and BLAS-like operations provided by the Intel MKL library are more efficient due to the optimal use 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 fare worse than the framework, which is able to map operations involving sub-matrices to optimized library calls.

GPU Performance

Similarly, for the GPU, we compare the performance of the OpenACC compiler generated code and the DTB-CUBLAS codes with the original un-annotated C code as the baseline. The parallelizable loops are annotated with the `acc kernels loop` directive, with an additional `independent` clause to indicate that there are no loop dependencies, and with the `collapse` clause for two-dimensional loops. Similarly, the reduction loops are also annotated. The OpenACC codes are compiled using the PGI compiler (version 13.10). The GPU codes are executed on the Nvidia Tesla

K20 GPU and the results are also presented in Table 6.6.

In most of the benchmarks, DTB-CUBLAS shows either comparable or better performance compared to the OpenACC compiled codes. This is because DTB-CUBLAS is able to recognize the linear algebra operations in the code and map them to optimized CUBLAS calls which make better use of architectural features such as software-managed cache. On the other hand, the OpenACC compiled codes are 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 sub-arrays 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.

6.3 Summary

We evaluate the effectiveness of our code transformation framework on six different benchmarks for multiple platforms. The first Section 6.1 shows the performance for the linear algebra template libraries (Eigen, Armadillo-MKL and Armadillo-CUBLAS), including the adaptive mapping using the cost model from Section 6.1.3. We see that by using the correct cost model which is tuned for the underlying platform, the benchmarks are able to adapt to the different architectures and provide near-optimal performance under widely differing scales. Section 6.2 illustrates the Direct to BLAS (DTB) component, targeting the following platform-optimized BLAS libraries: Intel MKL and Nvidia CUBLAS. We show that better performance can be achieved due to the use of these hardware-specific optimized libraries as compared to simple compiler directive annotated codes.

Chapter 7

Comparisons with Related Work

We have taken an unique approach to assist the user in getting better performance for their application on multiple platforms through linear algebra operations. In this chapter, we compare our approach with some other frameworks.

7.1 Source-to-source Transformation

There are a few source-to-source transformation frameworks that target parallelization. ROSE [29] is a compiler infrastructure for building source-to-source transformations for C/C++ and Fortran applications. Liao and coworkers demonstrate an approach that built upon ROSE to recognize high-level programming constructs such as the STL vector type [30]. By making use of the semantics of the high-level constructs, they then invoke a semantic-aware parallelizer to automatically parallelize target loops or functions using OpenMP directives.

Similarly, the Cetus tool [31] collects array access-related and loop-related information, and performs data-dependence analysis on the information in order to automatically parallelize the loops using OpenMP directives. Another related work is the Extract Kernel tool proposed by Damevski [32]. This tool transforms a loop written in C into a CUDA kernel function for execution on a GPU. Only loops which satisfy a set of preconditions are transformed. In the tool, kernels are launched by manually specifying the grid size and block size.

	Input	Output	
ROSE	STL Containers	OpenMP	
Cetus	Loops		OpenMP
Extract Kernel Tool	Loops		CUDA
Our Framework	Loops	Linear Algebra Libraries	

Table 7.1: Comparison for Source-to-Source Transformation Tools

These tools transform naïve codes to parallel codes. The transformation often depends on the platform that the parallel codes are targeting. Our approach differs in that we focus on identifying linear algebra operations, instead of just targeting parallelizable loops. In doing so, it will be easier to transform them to optimized library code based on their semantics. Table 7.1 shows a short comparison on the language features that are transformed and the resultant output from each tool.

7.2 Adaptive Code Generation

In the area of adaptive code generation, automatic algorithmic composition is a common approach, and has been applied in several auto-tuned software packages, such as ATLAS [33] and FFTW [34]. These software packages contain training phases where the optimal algorithms are automatically tuned and generated. Nevertheless, they are limited by the few algorithms provided by the library designers, often targeting only a single platform.

Another development is PetaBricks [35], an implicitly parallel language and compiler where the user is able specify multiple implementations of an algorithm. The PetaBricks compiler then auto-tunes programs by making fine-grained algorithmic choices, given the user-specified algorithmic implementations. Another related work is Qilin [36], a programming system that adaptively maps algorithms to the CPU and GPU during runtime. To achieve this, the user writes different algorithms for the CPU and GPU using the Qilin Application Programming Interface (API).

These systems require the user to implement the algorithms using a proprietary language or API whereas our framework only requires the user to provide a single sequential code, which can be transformed to linear algebra operations. We provide

	Adaptive Code	Auto-Tune
PetaBricks	Alternative Functions	Offline
Qilin	Alternative Functions	Online
SPIRAL	Generate from DSL	Online
Delite	Generate from DSL	Online
Our Framework	Generate from Linear Algebra	Offline

Table 7.2: Comparison for Adaptive Code Generation Frameworks

an adaptive mapping scheme for selecting the optimal platform for a particular problem size during runtime, based on the database that is constructed offline.

The SPIRAL program generation system [37] adopts a different approach. It 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 [38] targets heterogeneous execution by providing a framework for expressing algorithms through the use of an embedded domain-specific language. A sequence of well-defined intermediate representations can be used by algorithm writers to expose the parallelism in their algorithms, and different implementations can be generated for different targets, such as Scala, C++ or CUDA. The decision to execute which variant of an implementation is deferred to runtime according to the available resources.

In the systems above, the user is required to use a domain-specific language to achieve adaptation for different platforms. Our framework is similar in a way, in that the user can write in a domain specific language, such as using the Linear Algebra Language (LAL). However, our approach performs the adaptation on the linear algebra operations, which are either extracted from the naïve codes, or specified in LAL. These other systems adapt at a function level based on the user definition. A summary of the comparison of the adaptive frameworks is shown in Table 7.2.

7.3 Optimization

The Tensor Contraction Engine (TCE) [39] automatically generates codes from its own domain-specific language based on tensors for different parallel platforms, including multi-core processors and GPU. It performs optimization on the tensor expression tree to reduce the number of operations and fusion to minimize the space required for temporaries. After optimization on the expression tree, it distributes and partitions the data across the machine, performs space-time transformation to trade storage for re-computation of certain quantities and tiling to improve CPU cache usage.

MATLAB Execution on GPU-based Heterogeneous Architectures (Megha) [40] maps the control flow dominated regions of the MATLAB code to the CPU and the data parallel regions to the GPU to improve the program performance. First, the code is simplified into single static assignment form (SSA)[22] and the type and shape of the variables are inferred from the program. SSA is a variation on the definition-use graph with the following properties:

1. each assignment creates a different variable name
2. at points where control flow joins, a special operation is inserted to merge different versions of the same variable.

The data parallel statements are identified and composed as a single kernel. Optimizations such as scalarization and index array elimination are applied to remove redundant arrays. Finally, the kernels are surrounded by loops and the loop order is reordered according to the hardware mapping. Scheduling is performed on the kernels to determine which kernel is executed on the GPU and when the data needs to be transferred into the GPU.

Intel’s Array Building Blocks (ArBB) [41] is a retargetable dynamic compilation framework. ArBB features an embedded domain-specific language expressed in terms of arrays, which is dynamically compiled during runtime for both multi-threading and vectorization. It performs two types of high-level optimizations: (i)

	Intermediate Representation
TCE	Tensors
Megha	Parallel Loops
ArBB	Arrays
Our Framework	Matrices and Vectors

Table 7.3: Optimization on different intermediate representation.

classical compiler optimization such SSA transformation, copy propagation, dead code elimination, and constant folding, and (ii) optimization on low-level sub-primitives such as fusion. After that, low level optimization such as parallelization and vectorization is applied to the code.

The main difference with our framework is the intermediate representation to which the optimization is applied. Table 7.3 shows the list of the intermediate representations that each framework utilize. However, we draw inspirations from these frameworks on the optimizations that can be applied to the intermediate representation. We look at possible compiler optimizations to be applied on the linear algebra abstract syntax tree. For example, we perform operand fusion on the matrices to reduce the number of BLAS operations. We also utilize kernel composition for statements that cannot be extracted as BLAS operations. However since these statements are only a small subset of the codes that are transformed, we currently do not perform aggressive optimization on these generated kernels. These frameworks generate optimized codes from the intermediate representation, while our framework maps to optimized libraries.

7.4 Alternatives to BLAS

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 write in naïve codes with minimal annotations to be able to target multiple platforms. Furthermore, in order to benefit from platform-specific optimizations, our focus is on identifying and extracting well-defined linear algebra 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 [42]. 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 [43]. Build-to-order BLAS [44] and Design-by-transformation BLAS [45] approach the problem from a different angle. Their goal is to generate optimized and tuned BLAS-like functions from high level kernel specifications.

7.5 Summary

Our code recognition and generation framework differs from these works in a few ways. First, BLAS represents only a subset of the possible linear algebra operations which our framework can handle. Hence, once BLAS extensions are available, it is relatively straightforward to extend our framework to provide support for them. Second, these other 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 libraries that are provided. For example, our framework can be extended to make use of other BLAS packages such as PLASMA or MAGMA [46]. The former targets multi-core systems whereas the latter targets heterogeneous architectures.

Chapter 8

Conclusion

In this chapter, we summarize the work described in this thesis and subsequently discuss possible directions for future work.

8.1 Summary

In this thesis, we describe a code transformation framework that can recognize and extract code which resembles linear algebra forms, and map them to optimized vendor libraries. Although users often write code to target different hardware platforms, many codes contain idioms which have equivalents in highly optimized vendor libraries. Therefore, we propose an approach to extract known idioms and replace them with optimized library calls.

In order to achieve this, the framework first pre-processes the input source code. The domain for each loop is extracted and used to identify the shape of the arrays in the statement. There are three different techniques for the domain extraction:

1. Loop-based Extraction
2. User Annotated Loop Extraction
3. Unified Polyhedral Extraction

Additional preprocessing is done to reduce program variations to allow for better

pattern matching. After the preprocessing stage, the code undergoes a series of pattern-matching rules to identify the linear algebra operations in the code.

The linear algebra operations are output as an intermediate representation using the domain specific language, Linear Algebra Language (LAL). Users can also write their kernels using LAL directly. This intermediate representation is parsed and processed by the code transformation component.

From the linear algebra intermediate representation, the framework transforms the linear algebra operations to the equivalent libraries that are highly tuned for the underlying platform. The transformation component maps the linear algebra operations to either the Linear Algebra Template Libraries or directly to BLAS function calls.

The mapping of the hardware platform to the Linear Algebra Template Libraries:

Vector Processing Unit Eigen C++ Template Library

Multi-core Processor Armadillo Template Library linked with Intel MKL

GPU Armadillo Template Library linked with Nvidia CUBLAS

Because of the equivalence of the mathematical operations, we are able to substitute different library routines for the same operation. Since no one library is most effective under all circumstances, we extend the framework to pick the most effective one for the given local conditions. At the same time, different parts of the same application can be executed on different platforms depending on which is the most suitable for the given abstraction, thereby ensuring the best overall performance.

However there are disadvantages of using linear algebra template libraries. It incurs additional performance overhead due to code expanding by the compiler. To remove the dependencies and overhead on the template libraries, the framework is extended to generate code for Intel MKL or Nvidia CUBLAS directly. This transformation component also substitutes operations with our internal implementation if there is no function available for the platform.

We demonstrate and evaluate the extraction and transformation process in two different experiments. The first experiment evaluates the performance of our transformation framework targeting linear algebra template libraries (Eigen, Armadillo-MKL and Armadillo-CUBLAS). Our experiments show that using the appropriate cost model, our framework is able to target different levels of parallelism most suitable for the problem size. In the second experiment, we show that better performance can be achieved due to the use of hardware-specific optimized libraries (Intel MKL and Nvidia CUBLAS) as compared to compiler directive codes. 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.

8.2 Future Work

MATLAB [47], an array language, is popular for implementing numerical and scientific applications. MATLAB programs naturally express data-level parallelism, but do not offer high performance. It is possible to use GPUs for MATLAB programs, but it requires the user to manually rewrite the program to support GPU execution. Therefore it is useful if the framework is able to parse MATLAB programs and automatically generate efficient codes to run on different hardware.

Currently only general square matrices are recognized. However, in the future, the implementation can be extended for other types of matrices such as symmetric, triangular and Hermitian. Extending the support to other types of matrices will allow the framework to support a larger set of linear algebra algorithms, and allow greater flexibility for the user to write their applications.

At this stage, only a single kernel function is supported in the framework. Implicit cases such as when the operand vector is actually generated and returned by another function, are not supported. Additional support can be added to either allow nesting of functions, or inline the nested functions. This will allow the user to write modular codes, instead of modifying their codes to fit the framework.

The modified Armadillo for CUBLAS is inefficient due to the non-optimized use of memory transfer for the GPU. Another useful direction of future work would be designing a template library for CUBLAS. Although there is an existing template library, the LA Library [48], it is no longer updated since 2010. The new library will be designed to access the GPU memory easily, similar to the Thrust [49] library for parallel algorithms and data structures on Nvidia GPU. However, unified memory is introduced in CUDA 6, which enables a simpler memory design with implicit data transfers.

Intel Many Integrated Core Architecture (MIC) is a new computational platform, attached to the current hardware as a co-processor. For example, Intel Xeon Phi Co-processor 3120P features 512-bit vector processing units with 57 processors. Intel MKL is also available for execution on the MIC. The framework can be extended easily in future to support generation of MIC kernels. This includes memory transfer directives for the MIC and using Intel MKL as the BLAS library.

Similarly, extending support for OpenCL is useful for executing linear algebra applications on the AMD GPU platforms. There are newer BLAS libraries written in OpenCL, such as clMAGMA [46] and clBLAS [50]. OpenCL also requires a set of initialization and memory transfer functions for their libraries.

Appendix A

Linear Algebra Language

In this appendix, the context-free syntax of LAL is described using the EBNF grammar. The lexical syntax of LAL is based on Scala, which can be found in [51]. In Section A.1, the basic and array data types are introduced. Section A.2 gives details of the expressions that are evaluated according to the order of precedence in Section A.2.5. The expressions can be used to construct statements as shown in Section A.3. Section A.4 describes a function that is built from a sequence of statements.

A.1 Types

Type ::=

Basic-Type |

Array-Type

Basic-Type ::=

Integer |

Long |

Float |

Double |

Boolean

Array-Type ::= Basic-Type Array-Size1 { Array-Size2 }

Array-Size ::= '[' SimpleExpr ']'

All variables in LAL are defined with a type. Basic type represents the basic data type supported by LAL. There are four different basic types: **Integer**, **Long**, **Float** and **Double**. The integral data types can be defined as signed or unsigned. **Integer** type holds a 32-bits integer value while **Long** type holds a 64-bits integer value. Values of type **Integer** are all integer numbers between -2^{31} and $2^{31} - 1$, inclusive. Values of type **Long** are all integer numbers between -2^{63} and $2^{63} - 1$, inclusive. The type **Float** consists of all IEEE 754 32-bit single-precision binary floating point values, whereas the type **Double** consists of all IEEE 754 64-bit double-precision binary floating point values. **Boolean** type represents the values of *true* and *false*.

There is also a compound type **Array**, which represents a collection of values defined by (i) the basic type of the array and (ii) the dimension of the collection. The elements are defined to be stored linearly in memory in a column-major order. A one-dimensional array is also referred to as a vector, and a two-dimensional array as a matrix.

A.2 Expressions

```
Expr ::=
    Simple-Expr |
    Array-Expr |
    Nary-Expr |
    Unary-Expr
```

```
Simple-Expr ::=
    Literal |
    Identifier
```

Expression is a combination of identifiers and operators according to the rules of precedence and of association. This section lists the different types of expression in LAL.

A.2.1 Literals

```
Literal ::=
    IntegerLiteral |
    FloatingPointLiteral |
    BooleanLiteral
```

`Literal` represents a fixed basic type value. The basic types supported are detailed in Section A.1.

A.2.2 Array Expression

Array-Expr ::= Identifier '(' Array-Accessor1 {',' Array-Accessor2 } ')'

Array-Accessor ::=

Simple-Expr |

Slice-Expr

Slice-Expr ::= Simple-Expr ':' Simple-Expr ':' Simple-Expr

The array expression represents the reference to an array. The array can be referenced either by individual elements or using slicing notation. Accessing individual elements in the array is achieved by passing a list of **Simple-Expr**, where each element in the list refers to different dimension in the array. For example, $A(i,j)$ accesses an element of array A at position (i,j) . The arrays can also be accessed using the slicing notation, which can be used to represent a slice of the array. The slicing notation (**Slice-Expr**) is represented using triple of **Simple-Expr**, representing a range of integer values. Each **Simple-Expr** refers to the start of the value range, end of the value range, and the stride of each element respectively.

A.2.3 N-ary Expression

Nary-Expr ::= Expr {op Expr}

Nary-Expr contains an operator on a list of two or more expressions. The **Nary-Expr** describes an operation on the terms with no particular order according to the operator. The N-ary operators supported are shown in Table A.1. Table A.2 shows the logical operators that are also supported.

Operations	Descriptions
*	Scaling
+	Addition
.*	Dot Product
**	Matrix Product
< o >	Outer Product

Table A.1: N-Ary Operations

Operations	Descriptions
<	Less Than
≤	Less Than or Equal
>	Greater Than
≥	Greater Than or Equal
≡	Equals
≠	Not Equals
∧	And
∨	Or

Table A.2: Logical Operations

A.2.4 Unary Expression

Unary-Expr ::=

op Expr |

op '(' Expr ')'

Unary-Expr contains an unary operator with a term. The unary operators supported are shown in Table A.3. LAL also supports functional mapping of standard mathematical operations. This includes mathematical operations such as trigono-

Operations	Descriptions
—	Negation of a expression.
<i>sum()</i>	A reduction summation on an array.
<i>prod()</i>	A product on the elements of an array.
<i>tran()</i>	Transposition of a matrix

Table A.3: Unary Operations

Operations	Descriptions
<i>cos</i>	Compute cosine
<i>sin</i>	Compute sine
<i>tan</i>	Compute tangent
<i>acos</i>	Compute arc cosine
<i>asin</i>	Compute arc sine
<i>atan</i>	Compute arc tangent
<i>exp</i>	Compute exponential function
<i>log</i>	Compute natural logarithm
<i>pow</i>	Raise to power
<i>ceil</i>	Round up value
<i>floor</i>	Round down value
<i>abs</i>	Compute absolute value

Table A.4: Other Operations and Functions

metric functions (*sin*, *cos*, *tan*, $\dots$), as well as other operations such as *abs* and *log*. These operations can be applied either on a scalar or array. The list of the operations is shown in Table A.4.

A.2.5 Order of Operation

The order of operation defines the order in which the expression will be evaluated.

The precedence level of operators is as follows:

1. Unary Operators
2. Dot Product, Outer Product, Matrix Product
3. Scaling
4. Plus
5. Logical Operators

A.3 Statements

Statement ::=

Assignment-Statement |
Declaration-Statement |
Conditional-Statement |
For-Statement |
Do-Statement |
While-Statement |
Block-Statement

The **Statement** is the smallest stand-alone component of LAL that can be evaluated independently. It can be constructed from one or more expressions. Different types of statement are described in this section.

A.3.1 Assignment Statement

Assignment-Expr ::=

Identifier '=' SimpleExpr |
Array-Expr '=' Expr

Assignment Statement assigns an expression on the right-hand side (RHS) to another expression on the left-hand side (LHS). The interpretation of an assignment to a variable $\mathbf{x} = \mathbf{e}$ depends on the definition of $\mathbf{x}$. If $\mathbf{x}$ denotes a scalar variable, then the assignment changes the current value of $\mathbf{x}$ to be the result of evaluating the expression $\mathbf{e}$. If $\mathbf{x}$ is an array expression, the assignment depends on the definition of the array expression. For an individual elemental reference in the array expression, it is assigned the result of evaluating the expression $\mathbf{e}$. Otherwise, all the elements in the slices of the array denoted by $\mathbf{x}$ are assigned $\mathbf{e}$. If $\mathbf{e}$ is a **Slice-Expr**, individual element in the array denoted by $\mathbf{x}$ is assigned to each element in the slice.

A.3.2 Declaration Statement

Declaration-Statement ::= 'var' Declarator

Declarator ::= Identifier ':' Type

Declaration Statement is a statement that contains a Declarator. Declarator specifies the unique identifier and data type for a variable.

Declarator consists of an identifier and its type. The declarators are placed at the beginning of the Function or as the function parameters. The information regarding the identifier and data type is parsed and used to construct the symbol table.

A.3.3 Conditional Statement

Conditional-Statement ::=

'if' '(' Expr ')'

Block-Statement

['else' Block-Statement]

Conditional-Statement is the if-then-else construct that controls the control flow of the program based on the expression specified in the condition. The conditional expression `if (e1) s1 else s2` chooses to evaluate one of statements `s1` and `s2`, depending on the value of `e1`. The condition `e1` is expected to conform to type `Boolean`. The conditional expression is evaluated by evaluating first `e1`. If this evaluates to true, the statement `s1` is evaluated, otherwise the statement `s2` is evaluated.

A.3.4 Statement Block

Block-Statement ::= '{' Statement1 { Statement2 } '}'

Statement Block is a compound statement that contains a sequence of statements. Evaluation of a statement sequence entails evaluation of the statements in the order they are written.

A.3.5 Loops

Classical loop structures such as **For**, **Do** or **While** are supported.

For-Statement ::=

‘for’ ‘(’ Declarator ‘,’ Expr ‘,’ Expr ‘)’ Block-Statement

For Statement is an iteration statement that repeatedly executes statements in **Block-Statement**. The header of the statement contains a declarator for a loop variable and the upper and lower bound expressions of the iteration.

Do-Statement ::= ‘do’ Block-Statement ‘while’ ‘(’ Expr ‘)’

While-Statement ::= ‘while’ ‘(’ Expr ‘)’ Block-Statement

Do or **While** are loop structures containing a **Block-Statement** with a conditional exit out of the loop. The conditional check for **Do-Statement** is at the end of the statement block, while it is at the beginning for **While-Statement**.

A.4 Function

Function:

```
'def' identifier '(' Parameter1 {',' Parameter2 } ')'  
    Block-Statement
```

Parameter ::=

```
[In] Declarator |  
[Out] Declarator |  
[InOut] Declarator
```

Function defines a kernel and contains a list of **Declarators** as function parameters, where each declarator can be defined as an input and/or output. The kernel does not contain a return type; any variables to be returned are to be defined as an output variable.

These kernels mirror the structure of GPU kernels, where values can only be transferred from the host to the GPU through memory pointers passed to the GPU kernel. Therefore during code generation, it will be simpler to generate codes to be run on GPUs.

Appendix B

Additional Results

In this appendix, we include additional results for the benchmarks shown in Figure 6.1 and 6.2. In Figure B.1 and B.2, we plot the speedup (> 1) or slowdown (< 1) achieved by each of the code generators over the original baseline implementation. In the figures, the results for different code generators are labelled ‘Eigen’, ‘Armadillo-MKL’, ‘Armadillo-CUBLAS’ to denote vector processing unit-only, multi-core processor-only, and GPU-only execution respectively, and ‘Adapt’ to denote the adaptive mapping.

Figure B.1: Experimental results for *cg*, *gmres* and *jacobi*.

Figure B.2: Experimental results for *matexp*, *mcl* and *qr*.

References

- [1] P. Gepner and M. F. Kowalik, “Multi-core processors: New way to achieve high system performance,” in *Proceedings of the International Symposium on Parallel Computing in Electrical Engineering*, 2006, pp. 9–13.
- [2] I. Buck, “GPU computing: Programming a massively parallel processor,” in *Proceedings of the International Symposium on Code Generation and Optimization*, 2007, pp. 17–17.
- [3] *Intel Math Kernel Library Reference Manual*, Accessed 2014. [Online]. Available: <http://www.intel.com/software/products/mkl/>
- [4] *Eigen C++ Template Library*, Accessed 2014. [Online]. Available: <http://eigen.tuxfamily.org>
- [5] *Nvidia CUBLAS Reference Manual*, Accessed 2014. [Online]. Available: <http://developer.nvidia.com/cublas>
- [6] *AMD Core Math Library Reference Manual*, Accessed 2014. [Online]. Available: <http://developer.amd.com/libraries/acml>
- [7] W. J. Tan, W. T. Tang, R. S. M. Goh, S. J. Turner, and W.-F. Wong, “A code generation framework for targeting optimized library calls for multiple platforms,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. PrePrints, 2014.
- [8] 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.
- [9] *GNU Compiler Collection*, Accessed 2014. [Online]. Available: <http://gcc.gnu.org>
- [10] *IBM XL Compiler*, Accessed 2014. [Online]. Available: <http://www.ibm.com/software/products/en/xlcpp-aix>
- [11] “Intel C++ Studio XE,” Accessed 2014, <https://software.intel.com/en-us/intel-parallel-studio-xe/>.
- [12] NVIDIA, “Kepler whitepaper,” 2012. [Online]. Available: <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>
- [13] 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.
- [14] *Cray Programs*, Accessed 2014. [Online]. Available: <http://www.cray.com/Programs.aspx>
- [15] *PGI Accelerator Compiler (The Portland Group)*, Accessed 2014. [Online]. Available: <http://www.pgroup.com/>
- [16] *CAPS Enterprise Compiler*, Accessed 2014. [Online]. Available: <http://www.caps-entreprise.com/products/caps-compilers/>
- [17] 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.
- [18] *PathScale ENZO Compiler Suite*, Accessed 2014. [Online]. Available: <http://www.pathscale.com/enzo>

- [19] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Society for Industrial and Applied Mathematics, 1999.
- [20] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.
- [21] C. Sanderson, "Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments," Tech. Report, NICTA, Australia, 2010.
- [22] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.
- [23] M. Belaoucha, D. Barthou, A. Eliche, and S.-A.-A. Touati, "Fadalib: an open source C++ library for fuzzy array dataflow analysis," *Procedia Computer Science*, vol. 1, no. 1, pp. 2075 – 2084, 2010.
- [24] P. Feautrier, "Array expansion," in *Proceedings of the 2nd International Conference Supercomputing*, 1988, pp. 429–441.
- [25] C. Ancourt and F. Irigoin, "Scanning Polyhedra with DO Loops," *SIGPLAN Not.*, vol. 26, no. 7, pp. 39–50, Apr. 1991.
- [26] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parelo, 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.
- [27] 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.

- [28] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.
- [29] D. J. Quinlan, “ROSE: Compiler support for object-oriented frameworks.” *Parallel Processing Letters*, pp. 215–226, 2000.
- [30] C. Liao, D. J. Quinlan, J. Willcock, and T. Panas, “Semantic-aware automatic parallelization of modern applications using high-level abstractions,” *International Journal of Parallel Programming*, vol. 38, no. 5-6, pp. 361–378, 2010.
- [31] H. Bae, L. Bachega, C. Dave, S.-I. Lee, S. Lee, S.-J. Min, R. Eigenmann, and S. Midkiff, “Cetus: A source-to-source compiler infrastructure for multicores,” in *Proceedings of the 14th International Workshop on Compilers for Parallel Computing*, 2009, pp. 36–42.
- [32] K. Damevski and M. Muralimanohar, “A refactoring tool to extract GPU kernels,” in *Proceedings of the 4th Workshop on Refactoring Tools*, 2011, pp. 29–32.
- [33] R. C. Whaley and J. J. Dongarra, “Automatically tuned linear algebra software,” in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, 1998, pp. 1–27.
- [34] M. Frigo and S. G. Johnson, “The design and implementation of FFTW3,” *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.
- [35] 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.
- [36] C.-K. Luk, S. Hong, and H. Kim, “Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping,” in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 45–55.

- [37] 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.
- [38] 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*, 2011, pp. 35–46.
- [39] E. A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, H. Krishnan, C. chung Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, and E. Sibiryakov, “Automatic code generation for many-body electronic structure methods: The tensor contraction engine,” *Molecular Physics*, 2006.
- [40] A. Prasad, J. Anantpur, and R. Govindarajan, “Automatic compilation of matlab programs for synergistic execution on heterogeneous processors,” *SIGPLAN Not.*, vol. 46, no. 6, pp. 152–163, Jun. 2011.
- [41] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang, “Intel’s array building blocks: A retargetable, dynamic compiler and embedded language,” in *Proceedings of the 2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 224–235.
- [42] F. G. Van Zee and R. van de Geijn, “BLIS: A Framework for Rapidly Instantiating BLAS Functionality,” *ACM Trans. Math. Softw.*, 2013.
- [43] 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.
- [44] 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.
- [45] 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.
- [46] 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. IOP Publishing, 2009, p. 012037.
- [47] *Mathworks Home Page*, Accessed 2014. [Online]. Available: <http://www.mathworks.com/>
- [48] J. Pittner, “LA library: C++ interface to BLAS, LAPACK and CUBLAS linear algebra libraries,” 2010. [Online]. Available: <http://www.pittnerovi.com/la/>
- [49] J. Hoberock and N. Bell, “Thrust: A parallel template library,” 2014. [Online]. Available: <http://thrust.github.io/>
- [50] K. Knox, “clMath: An open source BLAS and FFT library for OpenCL,” 2014. [Online]. Available: <https://github.com/clMathLibraries>
- [51] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, “The Scala language specification version 2.9,” 2014.