

MAP-i Doctoral Programme

Parallel Programming by Transformation

PhD Pre-Thesis

Rui Carlos Araújo Gonçalves

Supervisors:
João Luís Sobral
Don Batory

January 2011

Abstract

The complexity of hardware platforms available today is increasing. Complex memory hierarchies, multi-core processors, GPUs as coprocessors, etc. bring new challenges to software developers. Developing efficient applications to a specific platform becomes increasingly more difficult. Additionally, developers usually have to maintain versions optimized to different platforms available, and it is usually difficult to the average programmer to understand highly optimized software.

This research addresses these challenges using model-driven engineering, a programming methodology that deals with software complexity raising the level of abstraction in program development. As a first step, we will focus on a specific area of application, *Dense Linear Algebra*, with the goal to automate development of customized, but well-understood, dense linear algebra programs. At the same time, we expect that our approach will not be domain-specific. Work in other domains will help confirm the generality of our work.

This document presents an overview of the areas and the research questions we are addressing. It also presents some preliminary results.

Contents

1. Introduction	3
1.1. Research Questions and Goals	4
1.2. Working Method	5
2. Background	6
2.1. Model-Driven Engineering	6
2.2. Linear Algebra Computations	9
2.2.1. Matrix Classifications	9
2.2.2. Operations	10
2.2.3. Basic Linear Algebra Subprograms	11
2.2.4. Linear Algebra Package	11
2.2.5. FLAME	11
3. Methodology	16
3.1. Models	16
3.2. Transformations	18
3.2.1. Refinements	18
3.2.2. Optimizations	18
4. Proof of Concept: Dense Linear Algebra Computations	21
5. Tool Support	29
6. Related Work	31
6.1. Knowledge-Based Software Engineering	31
6.2. Dataflow Programming	31
6.3. Program Optimization	32
6.4. Parallel Programming	33
7. Proposed Work	36
8. Conclusion	38

Chapter 1

Introduction

The computational power provided by hardware platforms in the last decades have been increasing. Initially, this increase was achieved mainly through higher clock rates, but at some point it was necessary to build more complex architectures to get more powerful platforms. Memory hierarchies, non-uniform memory architectures (NUMA), multi-core processors, clusters, or graphics processing units (GPU) as coprocessors, are examples of features that allowed to increase the computational power provided by hardware.

However, these features / resources are not *free, i.e.*, in order to get benefits from them, the programmer has to be careful in the development of the application, and tune the application to use the available features. As Sutter mentioned, *the free lunch is over* [Sut05]. The programmer has to choose the algorithm that best fits the target platform, he has to prepare the application to use multiple cores / machines, and apply other specific optimizations for chosen platform. Different platforms expose different characteristics, and that means that the best algorithm, as well as the optimizations, is platform dependent [WD98, GH01, GvdG08], *i.e.*, we need to develop and maintain several different versions of an application for the different platforms. This problem becomes even more important, because usually there is no separation between platform specific and platform independent code, limiting the reusability and making maintenance harder. Moreover, the platforms are constantly evolving (e.g. hardware and software changes), thus requiring an adaptation of the application.

This new reality moves the burden of improving performance of applications from hardware manufactures to software developers. To get full advantage of the hardware, applications must be prepared for it. This is a complex task, usually reserved for experts in the application domain. Moreover, developers need to have deep knowledge about the platform. These

challenges are particularly noticeable in high-performance computing, due to the importance given to performance in this community.

A particular type of optimization, that is becoming more and more important, due to the proliferation of parallel hardware, is algorithm parallelization. With this optimization, we want to improve applications' performance making them able to execute several tasks at the same time. This type of optimization will receive special attention in this work.

To answer these challenges, as well as to handle the growing complexity of the applications, we need new approaches. *Model-driven engineering (MDE)* is a software development methodology that has been used to address the complexity of software systems. In this work, we will explore the use of MDE techniques in the development of high-performance applications. We start with a specific domain, *Dense Linear Algebra (DLA)*. This is a well-know and mature domain, that has always received the attention of researchers concerned with highly optimized applications. We will use high-level, platform independent models and transformations to encode the knowledge collected by domain experts. However, it is not our goal to conceive new algorithms or implementations, but to develop a systematic methodology to record and apply this knowledge to develop systems automatically. Afterward, we will apply the knowledge obtained in DLA to other domains.

1.1 Research Questions and Goals

The lack of structure that characterise the development of efficient programs in domains such as DLA, prevents non-expert to participate in the development process. The main questions that we are trying to answer with this work are:

1. Can we develop general notations / models to represent applications and to express the optimizations that allow non-experts to understand, build and maintain them?
2. Can we develop methodologies and tools that help non-experts to build and maintain optimized (namely parallel) versions of their applications for several platforms, and that automate some of the tedious tasks in the optimization process?

MDE has been successfully used to explain the design of applications in several domains. This thesis aims to continue this line of work with a new programming methodology for high-performance computing, promoting

incremental development of applications, where complex applications are built refining, composing, extending and optimizing simpler blocks. Specific research goals are:

1. Develop a high-level, platform independent model to define operations and the algorithms that implement these operations. This model should help non-experts to understand the algorithms. It should also be extensible, to admit new algorithms and optimizations.
2. Develop methodologies to map operations to specific platforms. Decisions such as the choice of the algorithm, optimizations, and parallelization should be supported by these methodologies. The methodologies should help non-experts to understand how algorithms are chosen, and which optimizations are applied.
3. Develop tools to support the proposed methodologies.

1.2 Working Method

This research work will be divided in two phases:

1. In the first phase we focus on a specific domain, DLA. We start by studying the area, and identifying an appropriate set of models to express the domain knowledge, namely operations, algorithms and optimizations. Later, we will develop transformations to map high-level models to efficient implementations. We will also develop tools to help in the process of building applications. Finally we will study the information that experts need to choose a specific algorithm for a particular situation, or to apply an optimization, in order to make the tools able to perform some of the steps automatically.
2. In the second phase we will try to extend the work developed in the first phase, in order to make it applicable to other domains. This means to study other domains, namely its basic components (if they exist), and how applications are built and optimized.

Chapter 2

Background

2.1 Model-Driven Engineering

MDE is a software development methodology that promotes the use of models to represent the knowledge about a system, and model transformations to develop software systems. It lets the developers focus on the domain concepts and abstractions, instead of implementations details, and relies on the use of systematic transformations to map the domain models to implementations.

A *model* is a simplified representation of a system. It abstracts the details of a system, making it easier to understand and manipulate, while keeping the ability to provide the stakeholders that are using the model the details about the system they need [BG01].

Selic [Sel03] lists five characteristics that a model should have:

Abstraction. It should be a simplified version of the system, that hides insignificant details (*e.g.*, technical details about languages or platforms), and allows the stakeholders to focus on the essential properties of the system.

Understandability. It should be intuitive and easy to understand by the stakeholders.

Accuracy. It should provide a precise representation of the system, giving the same answers to stakeholders that the system would give.

Predictiveness. It should provide the needed details about the system.

Economical. It should be cheaper to construct than the physical system.

Models conform to a *metamodel*, which defines the rules that the meta-model's instances should meet (namely the *syntax* and constraints). For example, the metamodel of a language is its grammar, and the metamodel of a XML document is its XML schema or DTD.

The modeling languages can be divided in two groups. *General purpose modeling languages (GPML)*, that try to give support for a wide variety of domains and that can be extended when they don't fit some particular need. In this group we have languages such as the *Unified Modeling Language (UML)*. On the other hand, we have *domain specific modeling languages (DSML)*, that are designed to support only the needs of a particular domain or system.

Model *transformations* convert one or more source models into one or more target models. They manipulate models in order to produce new artifacts (*e.g.*, code, documentation, unit tests), and allow the automation of recurring tasks in the development process.

There are common types of transformations. *Refinements* are transformations that add details to models without changing their semantics or abstract structure, and they can be used to transform abstract specifications in implementations. *Abstractions* do the opposite, *i.e.*, remove details from models. *Refactorings* are transformations that restructure models without changing their semantics. *Extensions* are transformations that add new behavior or features to models. The transformations may also be classified as *endogenous*, when both the source and the target models are instances of the same metamodel (*e.g.*, a code refactoring), or *exogenous*, when the source and the target models are instances of different metamodels (*e.g.*, the compilation of a program).

MDE is used for several purposes, bringing several benefits to software development. The most obvious is the abstraction they provide, essential to handle the increasing complexity of software systems. Providing simpler views of the systems, they become easier to understand and to reason about, or even to show their correction [BR09]. Models are closer to the domain, and use more intuitive notations, thus even stakeholders without Computer Science skills can participate in the development process. This can be particularly useful in requirements engineering, where we need a precise specification of the requirements, so that developers know exactly what they have to build (natural language is usually too ambiguous to this purpose), expressed in a notation that can be understood by system's users, so that they can validate the requirements. Being closer to the domain also

makes models more platform independent, increasing reusability and making easier to deploy the system in different platforms.

Models are flexible (particularly when using DSML), giving freedom to users to choose the information they want to express, and how the information should be organized. Users can also use different models and views to express different views of the system.

Models can be used to validate the system or to predict the system's behavior without having to support the cost of building the entire system, or the consequences of failures in the real systems, that may not be acceptable [IAB09]. They can be used to check for cryptographic properties [JÖ5, ZRU09], to detect concurrency problems [LWL08, SBL08], or to predict performance [BMI04], for example. This allows the detection of problems in early stages of the design process, where they are cheaper to fix [Sch06, SBL09].

Automation is another benefit MDE. It dramatically reduces the time needed to perform some tasks, and usually leads to higher quality results than when tasks are performed manually. There are several tasks of the development process that can be automated. Tools can be used to automatically analyse models and detect problems, and even to help the user to fix them [Egy07]. Models are also used to automate the generation of tests [AMS05, Weiß09, IAB09]. Code writing is probably the most expensive, tedious and error-prone task in software development. With MDE we can address this problem, building transformations that automatically generate the code (or at least part of it) from models.

Empirical studies have already shown the benefits of using modeling in software development [Tor04, ABHL06, NC09].

Some of these tasks (*e.g.*, validation) could also be done using only code. It is important to notice that code is also a model. However usually it is not the best model to work with, because of its complexity and its inability to store all the needed information. For example, code loses information about the algorithm, which may be useful if we want to change the implementation of the algorithm. The use of code *annotations* clearly shows the need to provide additional information, *i.e.*, the need to extend the (code) metamodel. Moreover, code is only available in late stages of development process, which compromises the early detection of problems in the system.

The use of MDE also presents challenges to developers. One of the biggest difficulties when using MDE is the lack of stable and mature tools. This is a very active field of research, and we are seeing tools that exist to help code development being adapted to support models (*e.g.*, version man-

agement [GKE09, GE10, Kön10], slicing [LKR10], refactorings [MCH10], generics support [dLG10]), as well as tools that address problems more specific from MDE world (*e.g.*, model migration [RHW⁺10], graphs layout [FvH10], development of graphical editors [KRA⁺10]). Standardization is another problem. DSMLs compromise the reuse of tools and methodologies, as well as interoperability. On the other hand, GPMLs are too complex for most of cases [FR07]. The generation of efficient code is also a challenge. However, as Selic noticed [Sel03], this was also a problem in the early days of compilers, but eventually they become able to produce code as good as the code that an expert would produce. So we have reasons to believe that, as tools become more mature, this concern will diminish.

2.2 Linear Algebra Computations

Several sciences and engineering domains face problems where they need to use linear algebra operations to solve them. Due to its importance, the linear algebra domain has received the attention of researchers, in order to develop efficient algorithms to solve problems such as systems of linear equations, linear least squares, eigenvalue, or singular value decomposition.

This is a mature and well understood domain, with *regular programs*.¹ Moreover, the *basic building blocks* of the domain were already identified, and efficient implementations of these blocks are provided by libraries. This will be the first domain studied in this work.

In this section we provide a brief overview of the area, introducing some definitions and common operations. Developers that need highly optimized software in this domain, usually recur to well known APIs / libraries, that will also be presented.

2.2.1 Matrix Classifications

We present some common classifications of matrices, that help to understand operations and algorithms of linear algebra.

Identity A square matrix A is an identity matrix if it has ones on the diagonal, and all other elements are zeros. The $n \times n$ identity matrix is usually denoted by I_n (or simply I when the size of the matrix is not relevant).

¹DLA programs are regular because (i) they rely on dense arrays as their main data structures (instead of pointer-based data structures, such as graphs), and (ii) the execution flow of the program is predictable without knowing the input values.

Triangular A matrix A is triangular if it has all elements above or below the diagonal equal to zero. It is called lower triangular if the zero elements are above the diagonal, and upper triangular if the zero elements are below the diagonal. If all elements on the diagonal are zeros, it is said strictly triangular. If all elements on the diagonal are ones, it is said unit triangular.

Symmetric A matrix A is symmetric if it is equal to its transpose, $A = A^T$.

Hermitian A matrix A is hermitian if it is equal to its conjugate transpose, $A = A^*$. If A contains only real numbers, it is hermitian if it is symmetric.

Positive Definite A $n \times n$ complex matrix A is positive definite if for all $v \neq 0 \in \mathbb{C}^n$, $vAv^* > 0$ (or $vAv^T > 0$, for $v \neq 0 \in \mathbb{R}^n$ if A is a real matrix).

Nonsingular A square matrix A is nonsingular if it is invertible, *i.e.*, if there is a matrix B such that $AB = BA = I$.

Orthogonal A matrix A is orthogonal if its inverse is equal to its transpose, $A^T A = AA^T = I$.

2.2.2 Operations

LU Factorization. A square matrix A can be decomposed into two matrices L , unit lower triangular, and U , upper triangular, such that $A = LU$. This process is called *LU factorization* (or *decomposition*).

It can be used to solve systems of linear equations. Given a system of the form $Ax = b$ (equivalent to $L(Ux) = b$), we can find x , first solving the system $Ly = b$, and then the system $Ux = y$. As L and U are triangular matrices, any of these systems is easy to solve.

Cholesky Factorization. A square matrix A , that is hermitian and positive definite, can be decomposed into LL^* , such that L is a lower triangular matrix with positive diagonal elements. This process is called *Cholesky factorization* (or *decomposition*).

As LU factorization, it can be used to solve systems of linear equations, providing a better performance. However, it is not as general as LU factorization, as the matrix has to have certain properties.

QR Factorization. A matrix A can be decomposed into QR , such that Q is an orthogonal matrix, and R is an upper triangular matrix. This process is called *QR Factorization* (or *decomposition*).

QR factorization can be used to solve the linear least squares problem.

2.2.3 Basic Linear Algebra Subprograms

Basic Linear Algebra Subprograms (BLAS) is a standard API for linear algebra domain, that provides basic operations over vectors and matrices [LHKK79, Don02a, Don02b].

The operations provided are divided in three groups. *Level 1* provides scalar and vector operations, *level 2* provides matrix-vector operations, and *level 3* matrix-matrix operations. These operations are the basic building blocks of the linear algebra domain, and upon them, we can build more complex programs.

There are several implementations of BLAS available, developed by the academic community and hardware vendors (such as Intel [Int] and AMD [ACM]), and optimized to different platforms. Using BLAS, the developers are released from having to optimize the basic functions for different platforms, contributing to better performance portability.

2.2.4 Linear Algebra Package

The *Linear Algebra Package (LAPACK)* [ABD⁺90] is a library that provides functions to solve systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems. It was built using BLAS, in order to provide performance portability.

ScaLAPACK and PLAPACK are two extensions to LAPACK, that provide distributed memory implementations of some of the functions of LAPACK.

2.2.5 FLAME

The *Formal Linear Algebra Methods Environment (FLAME)* [FLA] is a project that aims to make linear algebra computations a science that can be understood by non-expert in the domain, through the development of *a new notation for expressing algorithms, a methodology for systematic derivation of algorithms, Application Program Interfaces (APIs) for representing the algorithms in code, and tools for mechanical derivation, implementation and analysis of algorithms and implementations* [FLA]. This project

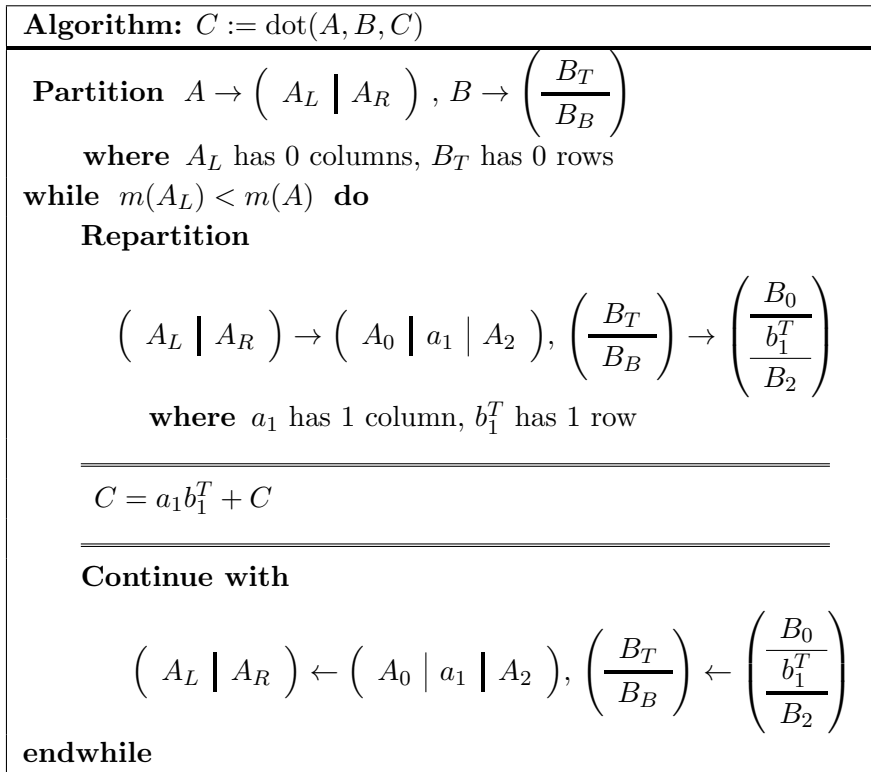


Figure 2.1: Matrix-Matrix Multiplication in FLAME Notation

also provides a library, *libflame* [ZCvdG⁺09], that implements some of the operations provided by BLAS and LAPACK.

The FLAME Notation. The FLAME notation [BvdG06] allows the specification of dense linear algebra algorithms without exposing the array's indices. The notation also allows the specification of different algorithms for the same operation, in a way that makes them easy to compare. Moreover, algorithms expressed using this notation can be easily translated to code using the FLAME API.

We show an example using this notation. Figure 2.1 depicts a matrix-matrix multiplication algorithm using flame notation (the equivalent Matlab code is shown in Figure 2.2).

Instead of using indices, in FLAME notation we start dividing the matrices in two parts (*Partition* block). In the example, matrix A is divided in A_L (the left part of the matrix) and A_R (the right part of the matrix), and

```

function [C] = dot(A, B, CO) {
    C = CO;
    s = size(A,2);

    for i = 1:s
        C = C + A(:,i) * B(i,:);
    end
}

```

Figure 2.2: Matrix-Matrix Multiplication in Matlab

matrix B is divided in B_T (the top part) and B_B (the bottom part).² The matrices A_L and B_T will store the part of the matrices that were already used in the computation, therefore initially these two matrices are empty. Then we have the loop, that iterates over the matrices, while the size of matrix A_L (given by $m(A_L)$) is less than the size of A , *i.e.*, while there are elements of matrix A that have not been used in the computation yet. At each iteration, the first step is to expose the values that will be processed in the iteration. This is done in the *Repartition* block. From matrix A_L we create matrix A_0 . The matrix A_R is divided in two matrices, a_1 (the first column) and A_2 (the remaining columns). Thus, we exposed in a_1 the first column of A that has not been used in the computation. A similar operation is applied to matrices B_T and B_B to expose a row of B . Then we update the value of C (the result), using the exposed values. At the end of the iteration, in the *Continue with* block, the exposed matrices are joined with the parts of the matrices that contain the values already used in the computation (*i.e.*, a_1 is joined with A_0 and b_1^T is joined with B_0). Therefore, in the next iteration the next column / row will be exposed.

For efficiency reasons, matrix algorithms are usually implemented using blocked versions, where at each iteration we process several rows / columns instead of only one. A blocked version of the algorithm from Figure 2.1 is shown in Figure 2.3. Notice that the structure of the algorithm remains the same. When we repartition the matrix to expose the next columns / rows, instead of creating a column / row matrix, we create a matrix with several columns / rows. Using Matlab (Figure 2.4), the indices make code complex (and using language such as C, that does not provide powerful index notations, it would be even more difficult to understand the code).

²In this algorithm we dived the matrices in two parts. Other algorithms may require the matrices to be divided in four parts, top-left, top-right, bottom-left and bottom-right.

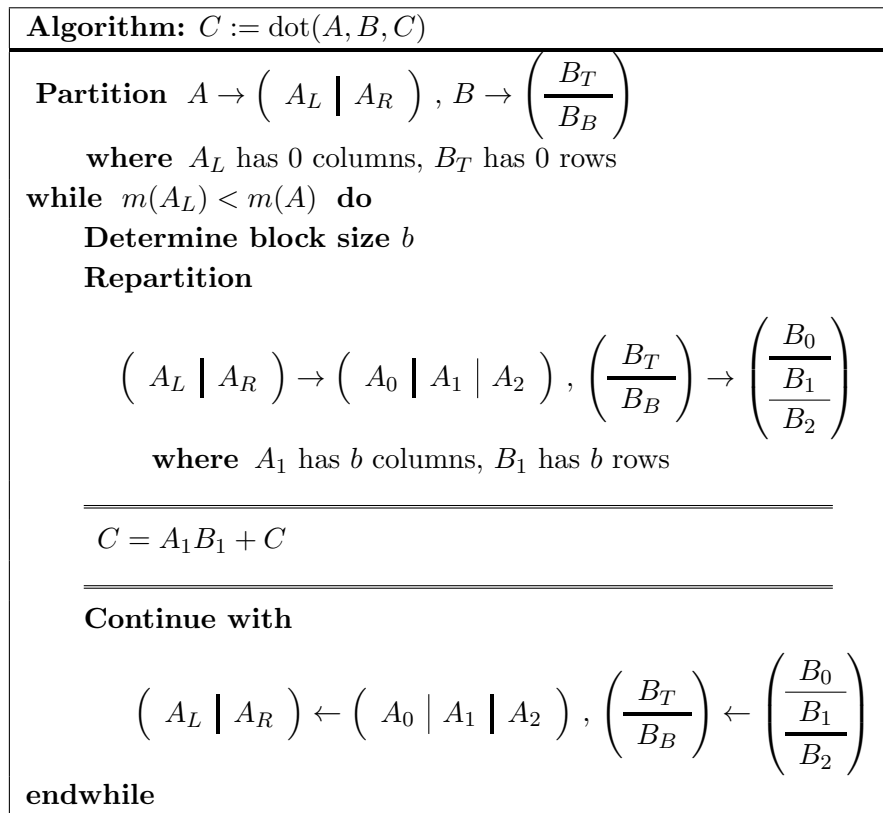


Figure 2.3: Matrix-Matrix Multiplication in FLAME Notation - Blocked Version

```
function [C] = dot(A, B, CO) {
    C = CO;
    s = size(A,2);

    for i = 1:mb:s
        b = min(mb, s-i+1);
        c = c + a(:,i:i+b-1) * b(i+b-1,:);
    end
}
```

Figure 2.4: Matrix-Matrix Multiplication in Matlab - Blocked Version

FLAME API. The *Partition*, *Repartition* and *Continue with* instructions are provided by FLAME API [BQOvdG05], which allows to easily translate an algorithm implemented in FLAME notation to code. The FLAME API is

```

function [ C_out ] = dot( A, B, C )
[ AL, AR ] = FLA_Part_1x2( A, ...
                          0, 'FLA_LEFT' );
[ BT, ...
  BB ] = FLA_Part_2x1( B, ...
                      0, 'FLA_TOP' );

while ( size( AL, 2 ) < size( A, 2 ) )
[ A0, a1, A2 ]= FLA_Repart_1x2_to_1x3( AL, AR, ...
                                       1, 'FLA_RIGHT' );

[ B0, ...
  b1t, ...
  B2 ] = FLA_Repart_2x1_to_3x1( BT, ...
                               BB, ...
                               1, 'FLA_BOTTOM' );

C = C + a1 * b1t;

[ AL, AR ] = FLA_Cont_with_1x3_to_1x2( A0, a1, A2, ...
                                       'FLA_LEFT' );

[ BT, ...
  BB ] = FLA_Cont_with_3x1_to_2x1( B0, ...
                                   b1t, ...
                                   B2, ...
                                   'FLA_TOP' );

end
C_out = C;
return

```

Figure 2.5: Matlab Implementation of Matrix-matrix Multiplication Using FLAME API

available for C and Matlab languages. The C API also provides some additional functions to create and destroy matrix objects, to obtain information about the matrix objects, and to show the matrix contents.

Figure 2.5 shows the implementation of matrix-matrix multiplication (unblocked version) in Matlab using the FLAME API (notice the similarities between this implementation and algorithm specification presented in Figure 2.1).

Chapter 3

Methodology

To address the challenges that software development in the high-performance computing community presents to developers, we use MDE. We defined high-level model to encode the domain knowledge, namely abstractions, algorithms and optimizations. Additionally, we specified transformations that allow developers to map high-level, platform independent program specifications to platform specific implementations. In this chapter we present the methodology that we use, namely the models and the transformations.

3.1 Models

An *abstraction* models an operation of the domain. It specifies its interface (input and output ports), and should also provide information about what it is supposed to do (*i.e.*, its semantics). Each port has a name (and may have other information, such as data type). In DLA, an abstraction can be a low-level BLAS routine, such as matrix-matrix multiplication (**GEMM**, **SYMM**, etc.), or more complex and high-level routines, such as Cholesky factorization.

Graphically, an abstraction is expressed by a box, with a name and input ports (on left) and output ports (on right). Figure 3.1 depicts an abstraction.

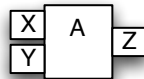


Figure 3.1: Graphical Representation of an Abstraction

An *algorithm* defines how an abstraction can be implemented. It has at least the same ports of the abstraction it implements (however it can have additional ports, that may be a parameter to tune performance, for

example). Additionally, an algorithm may contain abstractions and connectors (arrows between abstractions' ports), specifying which sequence of operations should be applied to input data to obtain the desired output. Alternatively, an algorithm can specify a code function that implements an abstraction.

Graphically, an algorithm is a box with ports, containing a dataflow graph. Figure 3.2 depicts an algorithm.

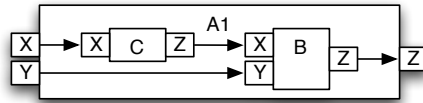


Figure 3.2: Graphical Representation of an Algorithm

A *grammar* pairs algorithms with the abstractions they implement. Each production of a grammar specifies an algorithm for an abstraction. The left-hand side is the abstraction, and the right-hand side is an algorithm. As textual grammars contain non-terminal, our algorithms contain abstractions. In textual grammars, productions define the valid replacements for non-terminal symbols. In our grammars, productions define the valid replacements for abstractions, *i.e.*, we can replace an abstraction with any of its algorithms. When replacing an abstraction by an algorithm we are adding detail to the model, thus we are performing a *refinement* of the abstraction.

The language defined by this grammar expresses all possible algorithms synthesizable by refinement of the domain.

Graphically, a grammar is specified by connecting algorithms to abstractions, as depicted in Figure 3.3.

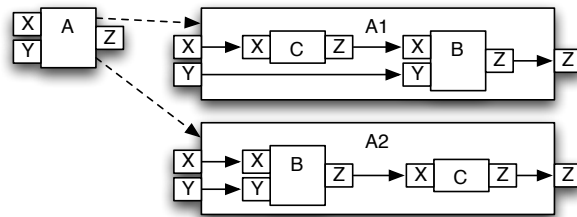


Figure 3.3: Graphical Representation of a Grammar: two productions showing two alternative implementations of abstraction A

3.2 Transformations

Given a *program design* (a dataflow model of a program), and a grammar of the abstractions and their implementing algorithms, we want to be able to build an efficient implementation of the program. As usual in MDE, we rely on model transformations (in this particular case, graph rewrites), to incrementally build a program. In the next sections we describe the two transformations we use to this purpose: refinements and optimizations.

3.2.1 Refinements

Given a program design, we can replace any of the abstractions present by one of its implementing algorithms. Doing this, we are adding details to the model of the abstraction (and, thus, to the model of the program), *i.e.*, we are refining the model.

Consider the grammar from Figure 3.3, and the program design from Figure 3.4a. Figure 3.4b shows a refinement of the program, where we replaced abstraction A with algorithm $A2$.

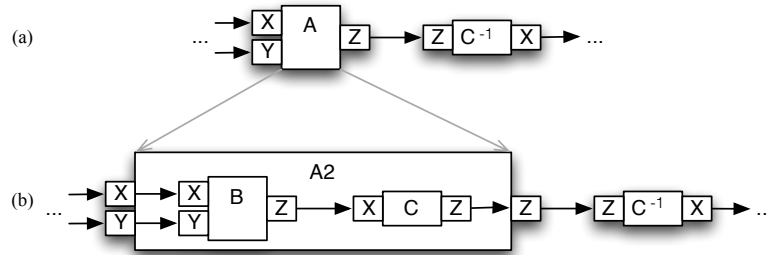


Figure 3.4: Refinement Transformation

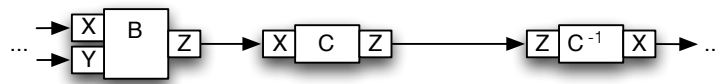
3.2.2 Optimizations

When we compose two dataflow graphs, we may get a sequence of operations that is inefficient. To address this issue, we provide optimizations. An *optimization* is a model transformation that consists on an abstraction transformation followed by a refinement. First the abstraction creates a single box representing the inefficient sequence of operations. Then we use a refinement to replace the created box with an efficient implementation.

Consider the program design, depicted in Figure 3.4b, that resulted from the refinement of abstraction A . We have the box C followed by box C^{-1} (that is the inverse operation of C). The output of box C^{-1} will be exactly

the input of box C , therefore these boxes can be removed. This will create a program that provides the same result, but that is more efficient.

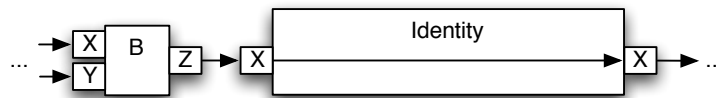
To remove these inefficiencies, we start by removing the encapsulation boundaries. Then we *abstract* the inefficient boxes to a single box. Finally, we *refine* the new box with an efficient implementation. Figure 3.5 shows the optimization process applied to the example presented. First, we remove box $A2$ (Figure 3.5a), then we abstract the sequence of boxes $C \rightarrow C^{-1}$, originating the box *Identity* (Figure 3.5b), and finally we refine the *Identity* box with an efficient implementation, that connects directly the input to the output of the box (Figure 3.5c).



(a) Step 1: remove $A2$ boundaries



(b) Step 2: abstract C and C^{-1} boxes



(c) Step 3: refine *Identity* abstraction

Figure 3.5: Design Optimization Steps

We also use the grammar to model the knowledge about optimizations. The subgraph that was abstracted is a possible implementation for the *Identity* abstraction, as well as the refinement applied in the last step. Thus, this optimization can be modeled using the grammar, as depicted in Figure 3.6.

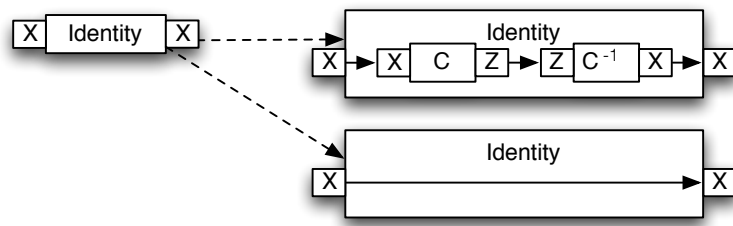


Figure 3.6: Model of an Optimization

Chapter 4

Proof of Concept: Dense Linear Algebra Computations

In this chapter we show an example where we synthesise a program using the approach we described. We synthesise a distributed memory implementation of Cholesky factorization, as described in [PMH⁺11]. The Cholesky algorithm used is depicted in Figure 4.1.

We have an iterative algorithm, that, at each iteration, updates some blocks of the matrix (A_{11} , A_{21} and A_{22}). To derive a distributed memory application, we will only model the part of the loop body that updates the matrix's blocks, as this is the only part of the algorithm that need to be refined. Figure 4.2 shows the initial program design of Cholesky loop body. The loop body has as input and output ports the three matrix blocks that are modified during an iteration.

We are deriving a distributed memory implementation of the Cholesky factorization, thus the input matrix blocks are distributed over several processors. To execute some operations we need to redistribute the elements of the matrix. Therefore, we have to refine the `Chol`, `Trsm` and `HerkLN` abstractions present in the initial design, with distributed memory implementations of them. Figure 4.3a and Figure 4.3b show the refinements and the resulting design, respectively. Abstractions of the format `xy-zw` (such that $x, y, z, w \in \{M, V, *\}$) are redistribution operations, that take a matrix using distribution `xy` and returns the matrix using distribution `zw`.¹

¹More details about these distributions and redistribution operations are provided in [PMH⁺11]. In abstractions names, we used `MM`, `M*`, `*M`, etc. to denote distributions $(\mathcal{M}_C, \mathcal{M}_R)$, $(\mathcal{M}_C, *)$, $(*, \mathcal{M}_R)$, etc.

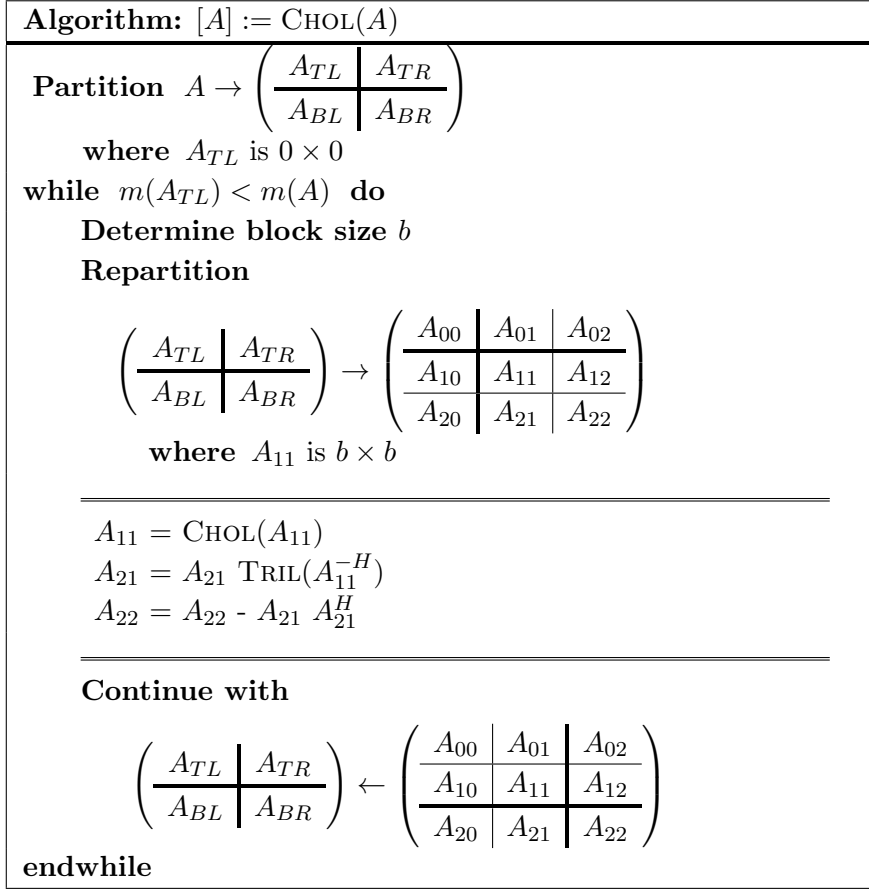


Figure 4.1: A Cholesky Algorithm

After flatten the new design, removing boxes `DChol`, `DTrsm` and `DHerKLN`, we can identify an opportunity for optimization (Figure 4.4a). We have the operation `**MM` followed by operation `MM**`, *i.e.*, we take a matrix using distribution $(*,*)$, we redistribute it to obtain the matrix using distribution $(\mathcal{M}_C, \mathcal{M}_R)$, and we redistribute it again to obtain the matrix using distribution $(*,*)$ (the final distribution is equal to the initial distribution). Figure 4.4b shows the model of the optimization. The algorithm `**+**+MM` (1) models the (inefficient) composition of boxes present in the current design, and that we abstract. The algorithm `**+**+MM` (2) is the efficient implementation of the same abstraction. After applying the optimization we obtain the design depicted in Figure 4.4c.

The next step is to refine some of the redistribution operations. This step is needed to expose new opportunities for optimization. We refine the

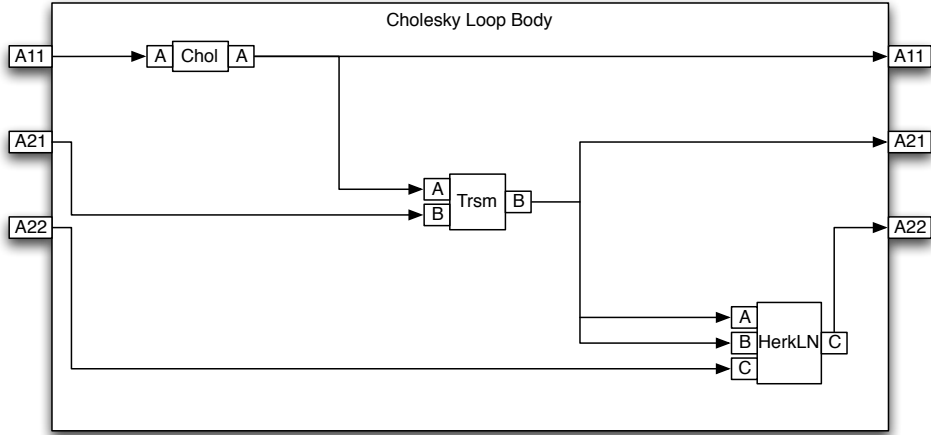


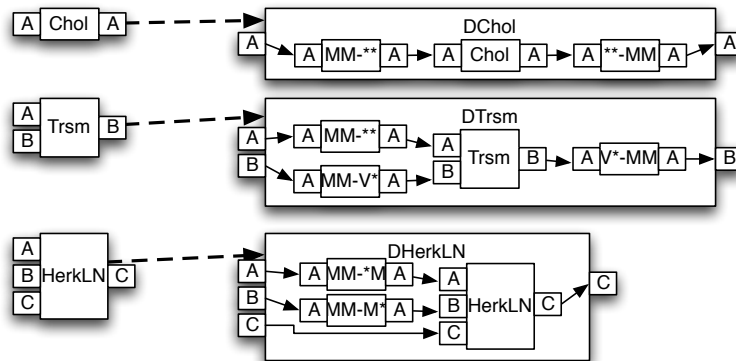
Figure 4.2: Initial Design of Cholesky Loop Body

redistribution operations $MM-M^*$ and $MM-*M$, using the algorithms presented in Figure 4.5a, and obtaining the design depicted in Figure 4.5b.

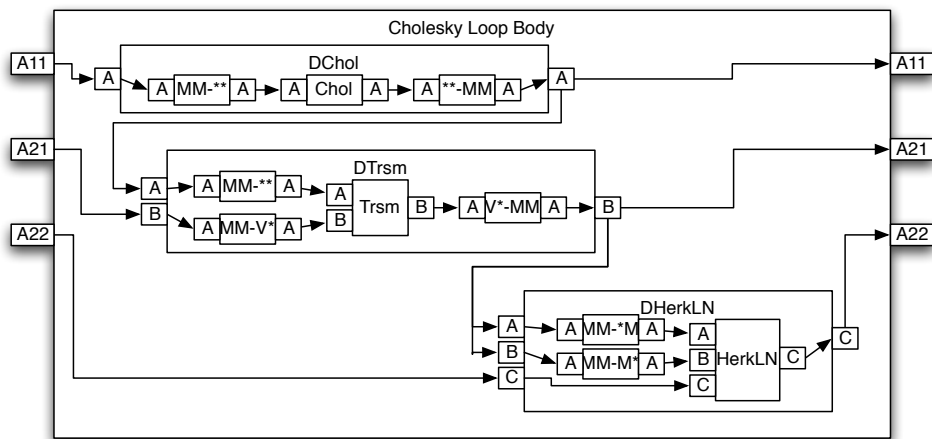
After flattening the new design, removing boxes $MM-V^*-M^*$ and $MM-V^*-*V-*M$, we can identify two optimizations (Figure 4.6a). We have an operation ($MM-V^*$) been applied twice to the same input. We can optimize this design applying the operation only once. We also have the sequence of operations $V^*-MM / MM-V^*$, that exposes an optimization similar to the first optimization we used. Figure 4.6b shows the models of the optimizations. Figure 4.6c shows the resulting design.

At this point we still have an optimization we can apply. We are using the redistribution operation V^*-MM to obtain a $(\mathcal{M}_C, \mathcal{M}_R)$ distribution of the output matrix of $Trsm$. Additionally, we have the redistribution operation V^*-M^* , that computes a $(\mathcal{M}_C, *)$ distribution of the same matrix. The $(\mathcal{M}_C, \mathcal{M}_R)$ distribution can be obtained more efficiently from a $(\mathcal{M}_C, *)$ distribution than from a $(\mathcal{V}_C, *)$, therefore we can apply the optimization depicted in Figure 4.7a. The final design is depicted in Figure 4.7b.

With this approach we were able to synthesise a distributed memory implementation of Cholesky factorization. We were able to use high-level models to encode the knowledge about the algorithms and optimizations needed to synthesise the application. Raising the level of abstraction, we can explain to non-expert how to derive efficient implementations of DLA programs, allowing them to participate in the development process. Notice that most of the optimizations used can be understood without any domain knowledge (in some cases, we only need to know that two boxes are the inverse



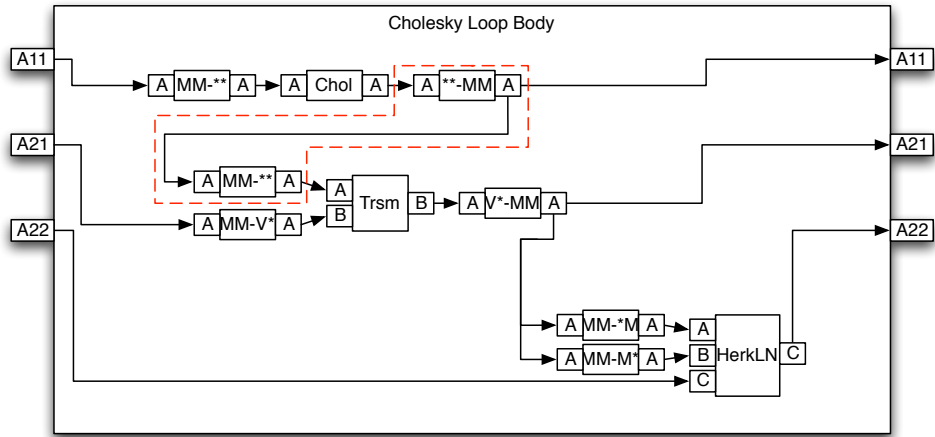
(a) Refinements' Models



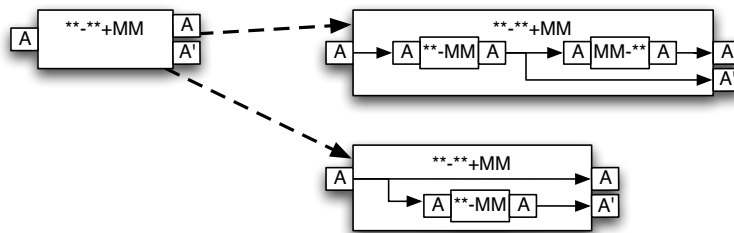
(b) Cholesky Loop Body After Refinements

Figure 4.3: Distributed Memory Refinements

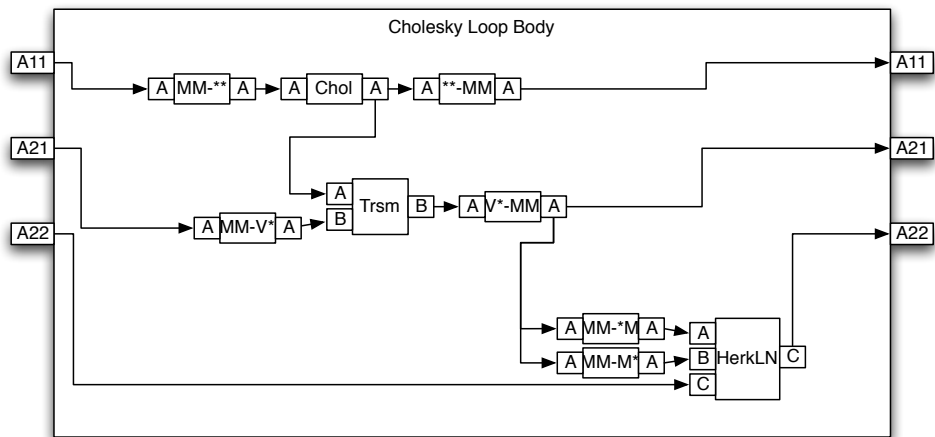
of each other, or to notice that the same box is used twice). Additionally, it also becomes easier to validate the program, as we only need to validate each of the refinements and optimizations. Finally, using a systematic approach to derive programs, we expect to enable synthesis automation.



(a) Redistribution Optimization Opportunity

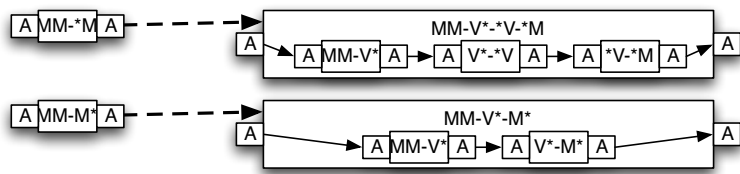


(b) Redistribution Optimization's Models

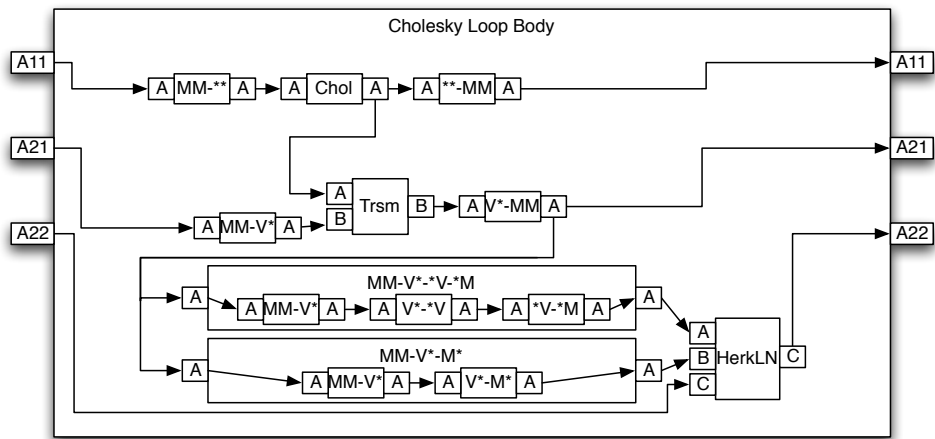


(c) Cholesky Loop Body After Optimization

Figure 4.4: Redistribution Optimization

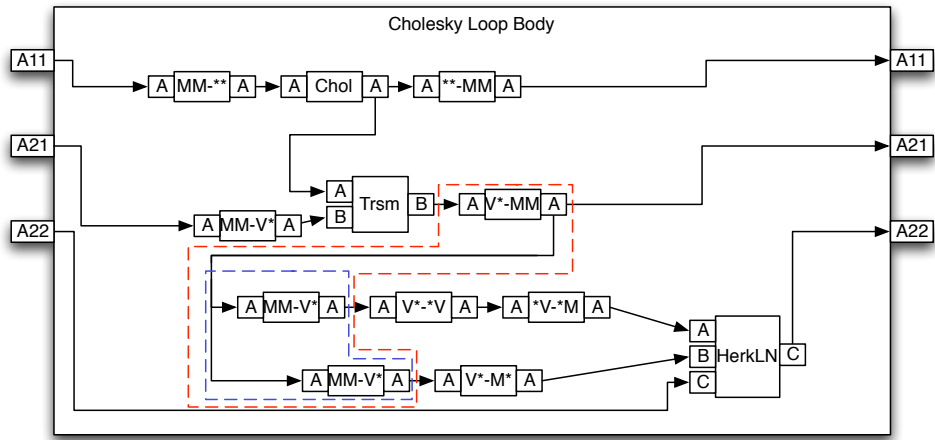


(a) Refinements' Models

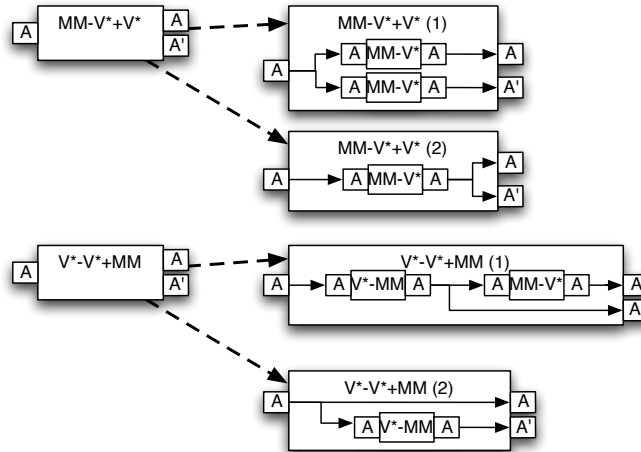


(b) Cholesky Loop Body After Refinements

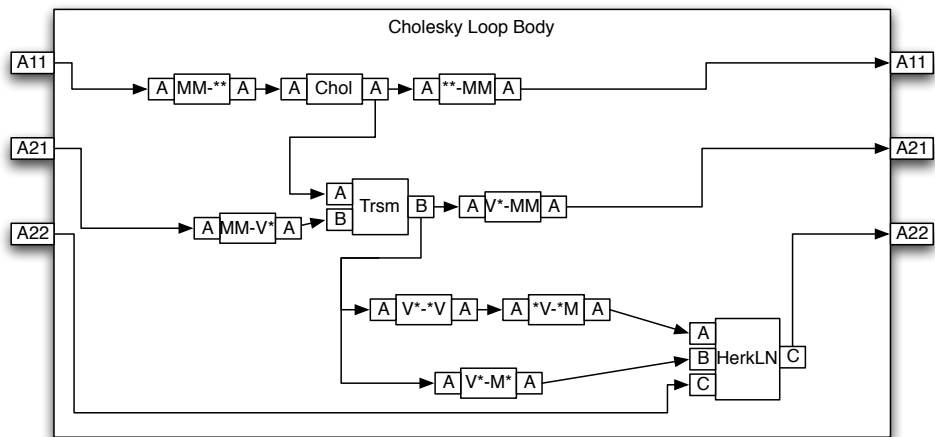
Figure 4.5: Redistribution Refinements



(a) Optimization Opportunities

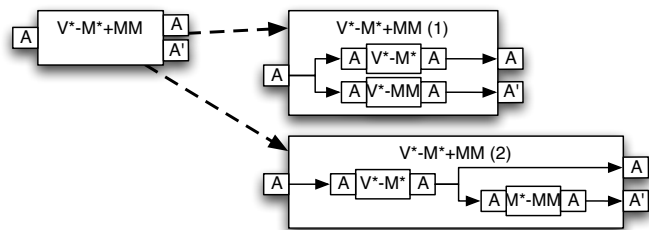


(b) Optimizations' Models

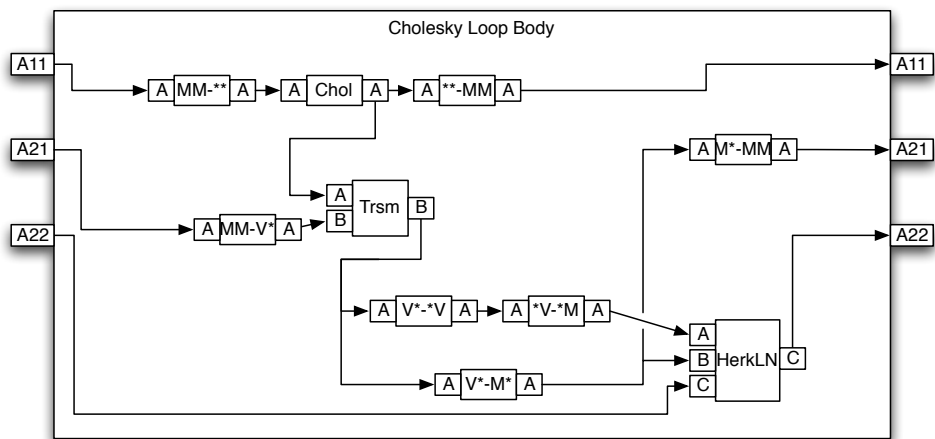


(c) Cholesky Loop Body After Optimization

Figure 4.6: Optimizations



(a) Optimization's Models



(b) Cholesky Loop Body After Optimization (Final Design)

Figure 4.7: Last Optimization and Final Design

Chapter 5

Tool Support

A tool to assist developers on the use of this approach is being developed. To use our approach, the developer starts encoding the domain knowledge, *i.e.*, specifying the domain abstractions and their implementing algorithms, as well as optimizations. To synthesise a program, the developer provides an initial design of the program, and then, refining abstractions and optimizing inefficient abstractions' compositions, he progressively maps the initial design to a design optimized to a specific platform. Finally, a transformation should be used to convert the final program design to code.

Our tool provides an environment for developers create models to encode domain knowledge, and to synthesise programs. Regarding models, it provides five types of objects: (1) *Abstraction*, (2) *Algorithm*, (3) *Pattern*, (4) *Input*, and (5) *Output*. The first three objects are boxes, and the last two are ports. Additionally, the tool has two types of associations: (1) *Connector* and (2) *Implementation*. Figure 5.1 shows a UML diagram of the tool's metamodel.

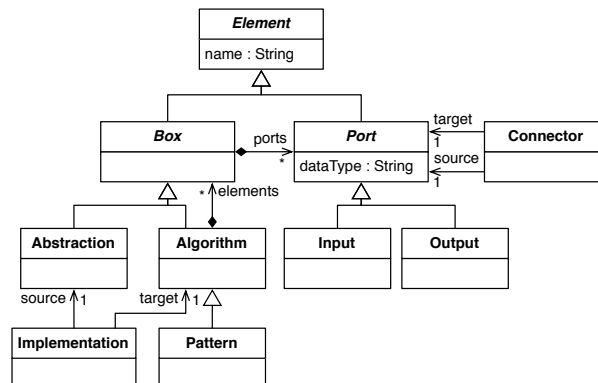


Figure 5.1: Tool's Metamodel

An *Abstraction* object with its input and output ports models an abstraction of the domain. An *Algorithm* object models an algorithm. In addition to its input and output ports, we add other boxes and *Connectors* inside, creating a dataflow graph that specifies an algorithm’s details. To build the grammar that pairs abstractions with their implementing algorithms, we use an *Implementation* association. A *Pattern* box specifies an algorithm to be abstracted. Like an algorithm box, it is paired with an abstraction and other plug-compatible algorithms. Patterns specify the algorithms that our tool will search for when optimizing designs.

These objects allow the developer to encode the domain knowledge. At this point, the developer can compose domain abstractions to build the initial design of the program he wants to synthesise.

Given an initial program design, the developer can replace any abstraction box with an algorithm that implements it. This can be done selecting the abstraction and applying the *refine* transformation. If there is more than one algorithm for the abstraction, the user will be asked to choose one. If a desired algorithm is not present, users can extend the list of algorithms and postulate his/her own designs.

Refinements produce a hierarchy of dataflow graphs that require optimizations. Our tool has a *flatten* transformation that removes modular boundaries. A user can then abstract a set of boxes and then refine to a new implementation. These two steps can be executed separately, using the *abstract* transformation, followed by the *refinement* transformation, or as a single step, using the *optimize* transformation. To help the user, we also provide the *find pattern* transformation, that identifies the compositions of boxes present in the program design that can be abstracted and then optimized.

Our tool performs several validations on models, namely to check if the dataflow graphs that specify the algorithms are valid, or if the algorithms have the same ports that abstractions they implement. In some cases, the tool can propose fixes to the errors, that the developer can select and automatically fix the errors.

Currently the synthesis process is done manually, *i.e.*, the user interactively selects the transformations to apply to the program design. We plan to enrich the models with additional information (*e.g.*, cost functions for algorithms) to allow the automation of this process. Our tool already allows us to obtain efficient designs, that only contain abstractions that can be directly mapped to code. However, this last step, of mapping a program design to code, is not yet implemented.

Chapter 6

Related Work

6.1 Knowledge-Based Software Engineering

Knowledge-Based Software Engineering (KBSE) was a methodology that emerged in the 1980s, and that promoted the use of transformations that mapped a specification to an efficient implementation [GLB⁺83, Bax93]. To develop a program, the developers would create a specification, and apply transformations to it, with the help of a tool, in order to obtain the implementation. Similarly, to maintain a program, the developers would only change the specification, and then they would replay the derivation process to get the implementation.

In KBSE, developers would work at specification level, *i.e.*, closer to the problem domain, instead of working at code level, where the knowledge about the problem is reduced, particularly when dealing with highly optimized code.

KBSE relied on the use of formal, machine-understandable languages to create specifications, and tools to mediate all steps in the development process. A tool would be used to record all developer's activities and the reason behind them, to apply transformations, and to help the developer choosing the transformations. The reliance on sophisticated tools and specification languages compromised its success [Bax93].

6.2 Dataflow Programming

Dataflow programming is a methodology that advocates the modeling of programs using directed graphs with data flowing between nodes [KM66, Den74, DK82]. The nodes express operations to be executed, and edges express the dependencies among them. Operations can be executed when

their input values are available, and several operations may be ready for execution at the same time, providing explicit parallelism.

This programming methodology is currently promoted by tools / languages such as LabVIEW [Lab], Simulink [Sim], or Agilent VEE [Agi].

6.3 Program Optimization

In this section we provide an overview of techniques that have been used to optimize software.

Peephole optimization [McK65] is an optimization technique that looks at a sequence of low-level instructions (this sequence is called the *peephole*, and its size is usually small), and tries to find an alternative set of instructions, that produces the same result, but that is more efficient. There are several optimizations this technique enables. For example, it can be used to compute expressions involving only constants in compile time, or to remove unnecessary operations, that sometimes result from the composition of high-level operations.

Loop transformations are optimization techniques that manipulate loop instructions in order to get more efficient code, namely improving data locality or exposing parallelism [PW86, WL91, WFW⁺94, AAL95, BDE⁺96]. Data layout transformations [AAL95, CL95] is another strategy that can be used to improve locality and parallelism. The success of these kind of techniques is limited for two reasons: the compiler only has access to the code, where most of the information about the algorithm was lost, and sometimes the algorithm used in the sequential code is not the best option for a parallel version of the application.

When using compilers or when using libraries, sometimes we have parameters we can set in order to improve performance. PHiPAC [BACD97] and ATLAS [WD98] address this question with parameterized code generators, that produce the different functions with different parameters, and time them in order to find out which parameters should be chosen for a specific platform. Yotov *et al.* [YLR⁺05] proposed an alternative approach, where although they still use code generators, they try to predict the best parameters using a model-driven approach, instead of timing the functions with different parameters. Several algorithms were proposed to estimate the optimal parameters [DS90, CM95, KCS⁺99].

Program transformations have been used to implement several optimizations in functional programming languages, such as function call inline, conditionals optimizations, reordering of instructions, function specialization,

or removal of intermediate data structures [JS98, Sve02, Voi02, Jon07]. Although this method is applied at higher levels of abstraction than loop transformations or peephole optimization, this approach offers limited support to developers specify new domain specific optimizations.

Rule-Based Query Optimization (RBQO) was an approach that allowed structure and reduce the complexity of query optimizers, and it was essential in the building of extensible database systems [Fre87, GD87, HFLP89]. Given a query, a query optimizer has to find a good *query evaluation plan (QEP)* that provides a strategy to obtain the results from the database system. In RBQO the possible optimizations are described by transformation rules, providing a high-level implementation independent, notation for this knowledge. In this way, the rules are separated from the optimization algorithms, increasing modularity and allowing incremental development of query optimizers, as new rules can be added, either to support more sophisticated optimizations or optimization for new features of the database, without changing the algorithms that apply the rules. The transformation rules specify equivalence between queries, *i.e.*, they say that a query that matches a pattern (and possibly some additional conditions), may be replaced by other query. Rules are also used to specify the valid implementations for query operators. Based on the knowledge stored in these rules, a rewrite engine produces several QEP. Different approaches can be used to choose the rules to apply at each moment, and to reduce the number of generated QEP, such as priorities attributed by the user [HFLP89], or the gains obtained in previous applications of the rules [GD87]. Later, cost functions are used to estimate the cost of each QEP, and the most efficient is chosen. This is probably the most successful example of the use of automatic optimization.

6.4 Parallel Programming

Several techniques have been proposed to overcome the challenges presented by parallel programming. One of the approaches that has been used is the development of languages with explicit support to parallelism. Co-array Fortran [NR98], Unified Parallel C (UPC) [CYZEG04] and Titanium [YSP⁺98] are extensions to Fortran, C and Java, respectively, that provide constructors for parallel programming. They follow the partitioned global address space (PGAS) model, that presents to the developer a single global address space, although it is logically divided among several processors, hiding communications from developers. Nevertheless, the developer still has to explicitly distribute the data, and assign work to each process. Mixing the parallel

constructors with the domain specific code, programs become difficult to maintain and evolve.

Z-level Programming Language (ZPL) [Sny99] is an array programming language. It supports the distribution of arrays among distributed memory machines, and provides implicit parallelism on the operations over distributed arrays. The operations that may require communications are, however, explicit, which allows the developer to reason about performance easily (WYSIWYG performance model [CCL⁺98]). However, this language can only explore data parallelism and when we use array based data structures. Chapel [CCZ07] is a new parallel programming languages, developed with the goal of improving productivity in the development of parallel applications. It provides high-level abstractions to support data-parallelism, task-parallelism, concurrency, and nested parallelism, as well as the ability to specify how data should be distributed. It tries to achieve a better portability avoiding assumptions about the architecture. Chapel is more general than ZPL, as it is not limited to data parallelism in arrays. However, the developer has to use language constructors to express more complex forms of parallelism or data distributions, mixing parallel constructors and domain specific code, and making programs difficult to maintain and evolve.

Intel Threading Building Blocks (TBB) [Rei07] is a library and framework that uses C++ templates to support parallelism. It provides high-level abstractions to encode common patterns of task parallelism, allowing the programmer to abstract the platform details. OpenMP [Boa08] is a standard for shared memory parallel programming in C, C++ and Fortran. It provides a set of compiler directives, library routines and variables to support parallel programming. It allows incremental development, as we can add parallelism to an application adding annotations to the source code, in some cases without need to change the original code. It provides high-level mechanisms to deal with scheduling, synchronization, or data sharing. These approaches are particularly suited for some well known patterns of parallelism (*e.g.*, the parallelization of a loop), but they offer limited support for more complex patterns, which requires considerable effort from the developer to explore them. Additionally, these technologies are limited to shared memory platforms.

Some frameworks take advantages of algorithm skeletons [Col91], that can express the structure of common patterns used in parallel programming. To obtain a program, this structure is parameterized by the developer with code that implements the domain functionality. A survey on the use of algorithm skeletons for parallel programming is presented in [GVL10]. Explore parallelism becomes particularly difficult in irregular applications, where we

may not know the tasks to be performed before the execution. Galois framework [KPW⁺07] addresses the problem of parallelize irregular applications rising the abstraction level of the applications' specification, and using optimistic parallelization techniques. These frameworks allow to rise the level of abstraction, and remove parallelization concerns from domain code. On the other hand, developers have to write the code according to rules imposed by frameworks, and using the abstractions provided by them.

One of the problems of parallel programming is the lack of modularity. In traditional approaches the domain code is usually mixed with parallelization concerns, and these concerns are spread among several modules (tangling and scattering in aspect-oriented terminology) [HG04]. Several works have used *Aspect-Oriented Programming (AOP)* [KLM⁺97] to address this problem. Some of them tried to provide general mechanism for parallel programming, for shared memory environments [CSM06], distributed memory environments [GS09], or grid environments [SGNS08]. Other works focused on solutions for particular applications [PRS10]. AOP can be used to map sequential code to parallel code without forcing the developer to write its code in a particular way. However, starting with a sequential implementation, the developer is not able to change the algorithm used. Additionally, AOP is limited regarding the transformations that it can make to code / programs. For example, it is difficult to use AOP to apply optimizations that break encapsulation boundaries.

All mentioned approaches tried to rise the level of abstraction at which developers work, hiding low-level details with more abstract language constructors or libraries. Nevertheless, the developer still has to work at code level. Additionally, none of the approaches allow the developer to easily change the algorithms, or provide high-level notations to specify domain specific optimizations.

Chapter 7

Proposed Work

We described a model-driven methodology to derive DLA programs. However, we are still far from the goals we defined. In this chapter we describe the remaining tasks.

DLA Algorithms (5 months)

We already showed how to use the proposed methodology to derive a program in DLA domain, for a particular platform (distributed memory machines). We will catalogue the most common operations of the domain and their implementing algorithms. We will apply the same methodology to derive other DLA operations, namely the operations presented in section 2.2.2.¹ Moreover, we will derive implementations to other platforms.

This may imply to extend the methodology with more powerful models (*e.g.*, currently we are only modeling loop bodies, but more complex optimizations may require to deal at the same time with operations inside and outside a loop body). We will also need other transformations, namely extensions.

Code Generation (3 months)

Currently we are already able to generate program designs were the operations present can be directly mapped to code implementations. However, the last step, of converting the model to code, still has to be done manually. We will enhance the models in order to allow the developers to specify how an operation should be mapped to code. Then we will develop a transformation to convert a program design to code.

¹We choose these particular operations because, according to domain experts, they are representative of the DLA domain.

Automatic Synthesis (7 months)

Automation is one of the most important benefits of MDE. In this research work, we plan to automate at least some of the steps of the synthesis process.

To achieve this goal, we will enhance the models, in order to store additional information such as properties about the data (*e.g.*, the structure of a matrix, or how a matrix is stored) and pre-conditions to algorithms (*e.g.*, the structure that a matrix must have so that a specific algorithm can be used), which will help to choose the best algorithms for a particular case. Moreover, we will add cost functions to algorithms, in order to allow performance prediction. This imply to work with different representations of an algorithm (*e.g.*, algorithm model, cost model), thus we need not only algorithm model transformations, but cost *models* – maybe other models – as well. As an algorithm’s performance depends on the hardware platform, we will also need platform models. Then we will explore algorithms to automatically generate different implementations, and to choose the most efficient for a specific platform.

The current graphical notation is appropriate for interactive program specification and synthesis. However, we plan to explore alternative notations, more amenable to automation.

Other Domains (5 months)

Currently this research is focused on the DLA domain. However, we already have evidence that this methodology can be applied to other domains. As last step, we will explore other domains, studying their basic components, algorithms and optimizations, and adapting the proposed methodology to them.

Chapter 8

Conclusion

In this document we proposed a metamodel to express the knowledge needed to build DLA programs. We identified key concepts of the domain, namely abstractions, algorithms, grammars and optimizations, and proposed models to express them. Additionally, we described the transformations that we need to synthesise efficient programs from the models. Using high-level models and graph rewrites, we were able to express algorithms and optimizations, as well as to replicate the process used by experts to derive efficient DLA programs in a systematic way. Moreover, we believe the notation and the transformations used are simple enough to allow non-experts in the domain to understand the development process. It also make design validation easier. We also presented a tool that we are developing to assist developers when using the proposed approach, and we successfully used the tool to synthesise a real program. Although we are focused on DLA domain, we already have evidence that our methodology can be applied to other domains (a similar methodology was applied to examples from database and fault-tolerance domains).

Several steps still need to be done to meet the goals we defined. We want to automate as much as possible the synthesis process. To do that, we are planing to enrich the model with properties about the input and pre-conditions on grammar's productions, that should help the choice of refinements to apply. Cost functions should also allow us to estimate the costs of different implementations of a program, and therefore to choose the best implementation. Additionally, we will also need to model hardware platforms, as the performance of a program, and thus the choice of the best program, depend on the platform. The final design obtained with the current tool only contain boxes that map directly to functions provided by external libraries such as BLAS and LAPACK, however we still need to implement

the transformation that generates the code from the final program design. The models and transformations we described already allowed us to derive some programs, but in other programs, such as LU factorization, we expect to need ability to support *extensions* (*i.e.*, the ability to add functionality to an abstraction). We plan to explore the use of alternative non-graphical notations, and tools to manipulate expressions in that notation, as an alternative to the current graphical models. Finally, we will study other domains and adapt the proposed approaches on them.

Bibliography

- [AAL95] Jennifer M. Anderson, Saman P. Amarasinghe, and Monica S. Lam. Data and computation transformations for multiprocessors. In *PPoPP '95: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 166–178, 1995.
- [ABD⁺90] E Anderson, Z Bai, Jack Dongarra, A Greenbaum, A McKenney, J Croz, S Hammerling, J Demmel, C Bischof, and D Sorensen. Lapack: A portable linear algebra library for high-performance computers. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 2–11, 1990.
- [ABHL06] Erik Arisholm, Lionel C. Briand, Siw Elisabeth Hove, and Yvan Labiche. The impact of uml documentation on software maintenance: An experimental evaluation. *IEEE Transactions on Software Engineering*, 32(6):365–381, 2006.
- [ACM] Amd core math library. <http://www.amd.com/acml>.
- [Agi] Agilent vee. <http://www.agilent.com/find/vee>.
- [AMS05] Mikhail Auguston, James Bret Michael, and Man-Tak Shing. Environment behavior models for scenario generation and testing automation. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–6, 2005.
- [BACD97] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 340–347, 1997.

- [Bax93] Ira Baxter. Practical issues in building knowledge-based code synthesis systems. In *Proceedings of the Sixth Annual Workshop on Software Reusability*, 1993.
- [BDE⁺96] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwenger, and Peng Tu. Parallel programming with polaris. *Computer*, 29(12):78–82, 1996.
- [BG01] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the omg/mda framework. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, 2001.
- [BMI04] Simonetta Balsamo, Antiniscia Di Marco, and Paola Inverardi. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.
- [Boa08] OpenMP Architecture Review Board. Openmp application 3program interface. <http://www.openmp.org/mp-documents/spec30.pdf>, May 2008.
- [BQOvdG05] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: the flame application program interfaces. *ACM Transactions on Mathematical Software*, 33(1):27–59, 2005.
- [BR09] Don Batory and Taylor L Riché. Stepwise development of streaming software architectures. Technical report, University of Texas at Austin, 2009.
- [BvdG06] Paolo Bientinesi and Robert van de Geijn. Representing dense linear algebra algorithms: A farewell to indices. Technical report, The University of Texas at Austin, Department of Computer Sciences, 2006.
- [CCL⁺98] B. Chamberlain, S. Choi, E. Lewis, C. Lin, L. Snyder, and W. Weathersby. Zpl’s wysiwyg performance model. In *HIPs '98: Proceedings of the High-Level Parallel Programming Models and Supportive Environments*, pages 50–61, 1998.

- [CCZ07] B L Chamberlain, D Callahan, and H P Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.
- [CL95] Michał Cierniak and Wei Li. Unifying data and control transformations for distributed shared-memory machines. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 205–217, 1995.
- [CM95] Stephanie Coleman and Kathryn S McKinley. Tile size selection using cache organization and data layout. *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming Language Design and Implementation*, pages 279–290, 1995.
- [Col91] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, 1991.
- [CSM06] Carlos A Cunha, João Luís Sobral, and Miguel Pessoa Monteiro. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 134–145, 2006.
- [CZEG04] François Cantonnet, Yiyi Yao, Mohamed Zahran, and Tarek A El-Ghazawi. Productivity analysis of the upc language. In *IPDPS '04: Proceedings of the 18th International Parallel and Distributed Processing Symposium*, pages 254–260, 2004.
- [Den74] Jack Dennis. First version of a data flow procedure language. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1974.
- [DK82] A. L. Davis and R. M. Keller. Data flow program graphs. *Computer*, 15(2):26–41, 1982.
- [dLG10] Juan de Lara and Esther Guerra. Generic meta-modelling with concepts, templates and mixin layers. In *MODELS '10: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems*, pages 16–30, 2010.

- [Don02a] Jack Dongarra. Basic linear algebra subprograms technical forum standard i. *International Journal of High Performance Applications and Supercomputing*, 16(1):1–111, 2002.
- [Don02b] Jack Dongarra. Basic linear algebra subprograms technical forum standard ii. *International Journal of High Performance Applications and Supercomputing*, 16(2):115–199, 2002.
- [DS90] Jack Dongarra and Robert Schreiber. Automatic blocking of nested loops. Technical report, University of Tennessee, Knoxville, TN, USA, 1990.
- [Egy07] Alexander Egyed. Fixing inconsistencies in uml design models. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 292–301, 2007.
- [FLA] Flamewiki. <http://z.cs.utexas.edu/wiki/flame.wiki/FrontPage>.
- [FR07] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *FOSE '07: Future of Software Engineering*, pages 37–54, 2007.
- [Fre87] Johann Christoph Freytag. A rule-based view of query optimization. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 173–180, 1987.
- [FvH10] Hauke Fuhrmann and Reinhard von Hanxleden. Taming graphical modeling. In *MODELS '10: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems*, pages 196–210, 2010.
- [GD87] Goetz Graefe and David J. DeWitt. The exodus optimizer generator. In *SIGMOD '87 Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 160–172, 1987.
- [GE10] Iris Groher and Alexander Egyed. Selective and consistent undoing of model changes. In *MODELS '10: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems*, pages 123–137, 2010.
- [GH01] Stefan Goedecker and Adolfo Hoisie. *Performance optimization of numerically intensive codes*. Society for Industrial Mathematics, 2001.

- [GKE09] Christian Gerth, Jochen M. Küster, and Gregor Engels. Language-independent change management of process models. In *MODELS '09: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 152–166, 2009.
- [GLB⁺83] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich. Report on a knowledge-based software assistant. Technical report, Kestrel Institute, 1983.
- [GS09] Rui Carlos Gonçalves and João Luís Sobral. Pluggable parallelisation. In *HPDC '09: Proceedings of the 18th ACM international symposium on High Performance Distributed Computing*, pages 11–20, 2009.
- [GvdG08] Kazushige Goto and Robert A van de Geijn. Anatomy of high-performance matrix multiplication. *Transactions on Mathematical Software*, 34(3), 2008.
- [GVL10] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.
- [HFLP89] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 377–388, 1989.
- [HG04] Bruno Harbulot and John R Gurd. Using aspectj to separate concerns in parallel scientific java code. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-Oriented Software Development*, pages 121–131, 2004.
- [IAB09] Muhammad Zohaib Iqbal, Andrea Arcuri, and Lionel Briand. Environment modeling with uml/marte to support black-box system testing for real-time embedded systems: Methodology and industrial case studies. In *MODELS '09: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 286–300, 2009.
- [Int] Intel math kernel library. <http://software.intel.com/en-us/articles/intel-mkl/>.

- [Jö5] Jan Jürjens. Sound methods and effective tools for model-based security engineering with uml. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 322–331, 2005.
- [Jon07] Simon Peyton Jones. Call-pattern specialisation for haskell programs. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 327–337, 2007.
- [JS98] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998.
- [KCS⁺99] Mahmut Kandemir, Alok Choudhary, Nagaraj Shenoy, Prithviraj Banerjee, and J. Ramanujam. A linear algebra framework for automatic determination of optimal data layouts. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):115–135, 1999.
- [KLM⁺97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, 1997.
- [KM66] Richard M. Karp and Raymond E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.
- [Kön10] Patrick Königmann. Capturing the intention of model changes. In *MODELS '10: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems*, pages 108–122, 2010.
- [KPW⁺07] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L Paul Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, 2007.
- [KRA⁺10] Dimitrios S. Kolovos, Louis M. Rose, Saad Bin Abid, Richard F. Paige, Fiona A. C. Polack, and Goetz Botterweck.

- Taming emf and gmf using model transformation. In *MODELS '10: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems*, pages 211–225, 2010.
- [Lab] Ni labview. <http://www.ni.com/labview/>.
- [LHKK79] C Lawson, R Hanson, D Kincaid, and F Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.
- [LKR10] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. Slicing of uml models using model transformations. In *MODELS '10: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems*, pages 228–242, 2010.
- [LWL08] Bin Lei, Linzhang Wang, and Xuandong Li. Uml activity diagram based testing of java concurrent programs for data race and inconsistency. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 200–209, 2008.
- [MCH10] Patrick Mäder and Jane Cleland-Huang. A visual traceability modeling language. In *MODELS '10: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems*, pages 226–240, 2010.
- [McK65] W. M. McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, 1965.
- [NC09] Ariadi Nugroho and Michel R. Chaudron. Evaluating the impact of uml modeling on software quality: An industrial case study. In *MODELS '09: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 181–195, 2009.
- [NR98] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.
- [PMH⁺11] Jack Poulson, Bryan Marker, Jeff R. Hammond, Nichols A. Romero, and Robert van de Geijn. Elemental: A new framework for distributed memory dense matrix computations.

- ACM Transactions on Mathematical Software*, 2011. Submitted.
- [PRS10] Jorge Pinho, Miguel Rocha, and João Luís Sobral. Pluggable parallelization of evolutionary algorithms applied to the optimization of biological processes. In *PDP '10: Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 395–402, 2010.
- [PW86] David Padua and Michael J Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–201, 1986.
- [Rei07] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., 2007.
- [RHW⁺10] Louis M. Rose, Markus Herrmannsdoerfer, James R. Williams, Dimitrios S. Kolovos, Kelly Garcés, Richard F. Paige, and Fiona A. C. Polack. A comparison of model migration tools. In *MODELS '10: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems*, pages 61–75, 2010.
- [SBL08] Marwa Shousha, Lionel Briand, and Yvan Labiche. A uml/spt model analysis methodology for concurrent systems based on genetic algorithms. In *MODELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 475–489, 2008.
- [SBL09] Marwa Shousha, Lionel C. Briand, and Yvan Labiche. A uml/marte model analysis method for detection of data races in concurrent systems. In *MODELS '09: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 47–61, 2009.
- [Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.
- [Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.
- [SGNS08] Edgar Sousa, Rui Carlos Gonçalves, Diogo Telmo Neves, and João Luís Sobral. Non-invasive gridification through an aspect-oriented approach. In *Ibergrid '08: Proceedings of the*

- 2nd Iberian Grid Infrastructure Conference*, pages 323–334, 2008.
- [Sim] Simulink - simulation and model-based design. <http://www.mathworks.com/products/simulink/>.
- [Sny99] Lawrence Snyder. *A programmer's guide to ZPL*. MIT Press, 1999.
- [Sut05] Herb Sutter. A Fundamental Turn Toward Concurrency in Software. *Dr. Dobbs's Journal*, 30(3):16–20, 2005.
- [Sve02] Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 124–132, 2002.
- [Tor04] Marco Torchiano. Empirical assessment of uml static object diagrams. In *IWPC '04: Proceedings of the 12th IEEE International Workshop on Program Comprehension*, volume 226–230, 2004.
- [Voi02] Janis Voigtländer. Concatenate, reverse and map vanish for free. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 14–25, 2002.
- [WD98] R Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27, 1998.
- [Weß09] Stephan Weißleder, Stephanisleder. Influencing factors in model-based testing with uml state machines: Report on an industrial cooperation. In *MODELS '09: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 211–225, 2009.
- [WFW⁺94] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary Hall, Monica Lam, and John Hennessy. Suif: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.

- [WL91] Michael Wolf and Monica Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452 – 471, 1991.
- [YLR⁺05] Kamen Yotov, Xiaoming Li, Gang Ren, María Jesús Garzaran, David Padua, Keshav Pingali, and Paul Stodghill. Is search really necessary to generate high-performance blas? *Proceedings of the IEEE*, 93(2):358 – 386, 2005.
- [YSP⁺98] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. In *Concurrency: Practice and Experience*, volume 10, pages 825–836, 1998.
- [ZCvdG⁺09] Field G Van Zee, Ernie Chan, Robert A van de Geijn, Enrique S Quintana-Orti, and Gregorio Quintana-Orti. The libflame library for dense matrix computations. *IEEE Design and Test*, 11(6):56–63, 2009.
- [ZRU09] M. Zulkernine, M. F. Raihan, and M. G. Uddin. Towards model-based automatic testing of attack scenarios. In *SAFE-COMP '09: Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security*, pages 229–242, 2009.