AUTOMATIC ALGORITHM DERIVATION AND EXPLORATION IN LINEAR ALGEBRA FOR PARALLELISM AND LOCALITY

BY

ALEXANDRE XAVIER DUCHÂTEAU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Abstract

Parallelization is one of the major challenges for programmers. But parallelizing existing code is a hard task that can lead to less than optimal solutions since sequential programs can suffer from impediments to parallelization resulting from the semantic of the languages or the data structures used rather than the nature of the problem being solved. To avoid such artifacts, programmers can analyze the algorithms to decide which dependencies are "real" and which can be ignored. But even then, conventional algorithms were developed with specific objectives in mind, such as reducing the total number of operations, which while good to achieve sequential performance, may not be the primary objective when considering parallel machines. We propose to focus on a specific domain and attack the parallelizing issue at the source, starting from a high level description of the equations without any knowledge of existing algorithms to solve the problem and automatically derive parallel solutions.

Hydra accepts an equation written in terms of operations on matrices and automatically produces highly efficient code to solve these equations. Processing of the equation starts by tiling the matrices. This transforms the equation into either a single new equation containing terms involving tiles or into multiple equations some of which can be solved in parallel with each other.

Hydra continues transforming the equations using tiling and seeking terms that Hydra knows how to compute or equations it knows how to solve. The end result is that by transforming the equations Hydra can produce multiple solvers with different locality behavior and/or different parallel execution profiles. Next, Hydra applies empirical search over this

space of possible solvers to identify the most efficient version. In this way, Hydra enables the automatic production of efficient solvers requiring very little or no coding at all and delivering performance approximating that of the highly tuned library routines such as Intels MKL.

With faster development time for modern architecture, the time available for hand-tuning of high performance libraries diminishes. Intel already started offering auto-tuned library routines (From Spiral) in their IPP [17] library, to broaden the scope of application of the collection, without having to increase the man hours required to hand-tune everything.

Acknowledgments

This work would not have been possible without the help and support of my advisors David Padua and Denis Barthou. I would also like to thank the members of my thesis committee Michael Heath, Sanjay Kale and María Garzarán for their feedback and encouragements.

I would like to give special thanks to Denis Barthou for getting me in computer science as an undergraduate student and following and encouraging me through my masters and then my thesis. As well as making the trip from France to attend the milestones of my thesis.

Throughout my undergraduate and graduate studies, many professors have been elemental in my interest in the field and fueled my passion for research, for that, William Jalby, David Kuck, Frank Capello, Euripides Montagne deserve my thanks.

My friends and colleagues both at the University of Illinois and back home were all instrumental in helping me settle in this new country. Special mention to Kelly Tang, Nik Dudukovic and the rest of the Illinois Underwater Hockey team that helped me relax and made my stay in Urbana Champaign a lot more enjoyable. And to Julien Jaeger who made the effort to read this thesis repeatedly to help me with editing.

Finally, I should not forget my family that unwaveringly supported me over the many years I spent on the path I chose to further my education. Making sure I had a home to come back to and forget about everything else for a blessed couple of weeks every year.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In theory there is no difference between practice and theory.

In practice, there is.

1.1 Background

With the fast development cycle of micro architectures, the task of producing and maintaining highly efficient code becomes increasingly difficult. With the added complexity of those architectures, the effort of hand tuning that is necessary to reach optimal performance is fast growing. In such an ecosystem, auto tuning solutions are particularly relevant. The intent behind such solutions and techniques is to move the development effort from the user to the machine, reducing the human hours spent in development to the cost of machine hours, which, with a proper balance, is cheaper and results in fewer human and overall hours. Auto tuning systems, are installed on a particular architecture, and given training time, produce a specialized implementation specific to the target architecture. This brings the additional advantage of making the auto tuned program portable through their capacity to adapt to a new architecture at installation. The major challenges attached to auto tuning are twofold, to define a comprehensive search space that contains optimal or near optimal solutions for the target architecture. And making sure that the complete exploration time required at installation is bound and kept within accepted reasonable time.

1.2 Overview

Years of research have led to very powerful algorithms to solve linear algebra on sequential machines. For parallel systems new techniques not needed in conventional systems must be developed to get the most of the available resources. Since both linear algebra algorithms and their implementations were initially developed with sequential machines in mind, existing algorithms or their implementations may not be the ideal place to start when looking towards parallel solutions to a problem.

Loss of information. Typically, a program is developed starting with an examination of the problem. Then, an algorithm to solve it is devised and refined with certain objectives in mind. We can assume that the goal was to minimize complexity while guaranteeing good numerical behavior. And while minimizing the complexity of an algorithm usually translates into less computation and thus faster sequential programs, this is not always the most important consideration for modern machines where locality and parallelism are of crucial importance. For parallel systems in particular, one should focus on minimizing execution time, reducing power consumption or a combination of these. Thus finding and exposing the independence of the computation becomes an important factor which is sometimes more important than minimizing the quantity of computation.

The next step is the implementation of the algorithm. Given that compilers often fail to generate optimal programs, programmers that aim at maximum performance will often apply transformations on their code to help the compiler in its optimization process to the point where it can become difficult to recognize what the code is doing. For example, figure 1.1a presents a simple triply nested loop that performs a matrix multiplication. Figure 1.1b is the same code, after application of a set of source optimization (tiling, variable promotion, unrolling, interchanging). The code now has three additional loops with larger strides, the innermost loop is now different and its body now presents two statements operating on scalars and single dimension arrays instead of three double dimension arrays and has two statements instead of one. While these transformations may optimize sequential performance on a specific machine, they may hide parallelism to a parallelizing or vectorizing compiler or even from a programmer.

```
  for(int i = 0 ; i < N ; i++)
    for(int j = 0 ; j < N ; j++)
      for(int k = 0 ; k < N ; k++)
        c[i][j] += a[i][k] * b[k][j];
```

(a) Matrix Multiplication Baseline

```
for(int ii = 0 ; ii < N ; ii+=B){
 for(int jj = 0 ; jj < N ; jj+=B){
  for(int kk = 0 ; kk < N ; kk+=B){
   for(int i = ii ; i < ii + B ; i++){
    for(int k = kk ; k < kk + B ; k++){
     c_i = c[i];
     a_ik = a[i][k];
     b_k = b[k];
     for(int j = jj ; j < jj+B ; j+=2){
      c_i[j] += a_ik * b_k[j];
      c_i[j+1] += a_ik * b_k[j+1];
     }
    }
   }
  }
 }
}
```

(b) Enhanced Matrix Multiplication

Figure 1.1: Two Possible Implementations for Matrix Multiplication

We thus make the argument that, when possible, parallel programs should be written starting at the problem specification rather than with a sequential implementation or algorithm.

Tuned parallel code generation. In this thesis, we propose to develop a system, for a class of linear algebra solvers, to automatically derive parallel algorithms from a high level description of the problem. This description includes the mathematical equation, and information on its operands. Working from this equation, the system will define parameters to characterize a class of parallel solutions using a divide and conquer approach, then explore this space of solutions to determine the best. Our system's output is a collection of equations connected by a dependence graph that describes a solution to the original equation.

1.3 Contribution

In this thesis, we will describe a system and framework to automatically produce parallel algorithms for a class of linear algebra solvers. To evaluate the approach, we developed a prototype and evaluated it on a set of problems.

Such a system could be used to extend mathematical interactive platforms such as matlab. Or by domain experts to directly generate high performance, portable, libraries for scientific and numerical approaches.

We start from a high level description of the target problem rather than with a known algorithm to solve it. The system then applies a collection of transformations to create different task dependence graphs describing parallel solutions.

The possible parallel algorithms can be evaluated using different metrics such as the size of the computational leaves, memory layout and locality, length of the critical path and maximum number of parallel tasks. When the target architecture is known, the machine's characteristics can be used to drive this selection and evaluation.

The dependence graphs can be used directly to devise algorithms or used as templates to generate parallel implementations. If the user provides a set of sequential kernels that solve the problem efficiently for known ranges of sizes, the system can generate task parallel code, for example in TBB [24, 16] or StarPU [3], and use user provided kernels for the computational nodes.

1.4 Thesis Organization

This Thesis is organized as follows. Chapter 1 introduced this work, Chapter 3 presents the system developed during this thesis while chapter 4 evaluates it. Chapter 5 presents possible extensions to this work for locality and widening its scope. Chapter 2 presents background and related work and Chapter 6 presents the conclusions.

Chapter 2

Related Work

In this chapter, we will present previous work done in the area of autotuning. First, section 2.1 will discuss some background work that inspired and lead to our own project. Then section 2.2 will present other projects directly related to our own. Finally, section 2.3 shows other projects in our field.

2.1 Background

The field of autotuning software generation tries to answer the problem of generating high performance libraries that are portable across platforms. The necessity comes from the fact that compilers often fail to produce the best possible executable from a normal source code, forcing programmers to manually develop codes that are only optimized for the specific machine it was developed for. Many projects have tackled this problem in different fields, proving the validity of exhaustive search to produce high performance library generators [6].

Yotov *et al.* [28] conducted a study to determine whether search was necessary or could be replaced by the use of theoretical models. Although they encountered some measure of success within the frame of ATLAS, they did not reject the necessity for search completely, especially in the case of algorithmic search [22, 11]. Moreover, models for new architectures are less reliable and parallel models even more unpredictable.

Previous work [4] on auto-tuning at source code level produce good results on matrix multiplication, but suffered on more complex problems. The intent behind xlanguage was to leverage state of the art compilers to breach the gap between hand tuned libraries coded directly in assembly code, and code written in higher level languages by normal developers. Doing so through the application of code transformations (scalar promotions, loop unrolling, tiling, etc.) at the source level rather than in the compiler. It was observed that compilers have very advanced analyses but are very complex pieces of software with heuristics that are very sensitive to the coding style. I.e. coding a same problem in slightly different ways could lead to widely different performance once compiled. This observation and further experimentations validated the idea of applying compiler transformations at the source level rather than only relying on the compiler to apply them. The project was successful in breaching this gap on BLAS operations on square matrices, though still producing slower programs than the hand tuned libraries. On the other hand, when looking corner cases not considered by the library vendors, for example, when operands were rectangular matrices, our automatic approach was able to exceed the performance of hand tuned libraries. The exploration space for source to source transformation has to be defined by the user through pragmas. For complex transformations, such as the ones leading to the task graphs produced by Hydra, the sequence of pragmas required would be difficult to identify, even by an expert. Besides, multiple implementations of a same algorithm can become radically different, advocating for looking at problems at a higher level.

2.2 Related Work

ATLAS [26] is a system that exhaustively searches a space of code transformations to find optimal implementations of matrix multiplications and other BLAS operations on a target machine. Later versions of ATLAS have combined performance prediction models as well as hand-coded kernels that can be used at the lower levels of blocking to improve the performance further.

The Spiral [22] project is the closest to what is proposed in this document. Spiral is a system to automatically generate high performance libraries for Digital Signal Processing (DSP). It offers a language and set of operators to specify linear transforms for DSP (Figure 2.1, from which their automatic generation system can derive different algorithms and in the end implementations. They also use search to evaluate performance and select the best implementation among all the versions generated by the system.

$$\begin{array}{l} \hline y = (A_n B_n)x \\ y = (I_m \otimes A_n)x \\ y = (A_m \otimes I_n)x \\ y = (\bigoplus_{i=0}^{m-1} A_n^i)x \\ y = D_{m,n}x \\ y = L_m^{mn}x \\ \hline \end{array}$$

(a) SPL Constructs

$$\mathrm{DFT}_{mn} \rightarrow (\mathrm{DFT}_m \otimes I_n) D_{m,n} (I_m \otimes \mathrm{DFT}_n) L_m^{mn}$$
$$\mathrm{DFT}_8 = (\mathrm{DFT}_2 \otimes I_4) D_{8,4} (I_2 \otimes (\mathrm{DFT}_2 \otimes I_2) D_{4,2} (I_2 \otimes \mathrm{DFT}_2) L_2^4) L_2^8$$

(b) FFT Described in SPL

Figure 2.1: Spiral SPL, Constructs and Examples

Our proposed system differs from Spiral by its targeted domain and from ATLAS in that it mainly focuses on exposing task parallelism. However, both projects offer insights in the different techniques that can be applied to guide the process of search through empirical execution of different implementations.

Kodukula and Pingali [19] introduced data centric transformations as a compiler optimization to reduce memory latency in memory bound codes. The approach proposed is similar to ours in that the optimizations are centered around the partitioning of the data rather than on the surrounding computation performed on it. But they work at the code level, partitioning data structures and restructuring loop nests while our approach intervenes at the algorithmic level when neither control flow nor data structures are present yet.

The Flame [5] project advocates goal-oriented programming. It offers a platform to develop algorithms in a systematic way. Flame offers a framework to write iterative algorithms, while we try to start from a problem and automatically derive algorithms recursively using a divide-and-conquer approach. Figure 2.2 shows an example of Flame program. Besides, the code generation approach presented here, relying on the dynamic scheduling of a parallel task graph, differs from the path chosen by Flame. Recent work from Fabregat-Traver and Bientinesi [10] proposes an approach close to ours for finding algorithmic solutions to matrix equations from their mathematical expression. However, they do not explain how the code is generated nor present any performance figures.

2.3 Other Work in the Field

The Plasma and Magma projects [13, 1] look at scheduling parallel task graphs on both multi CPU and multi GPU systems and has some encouraging results on heterogeneous systems. However, at this point they do not consider task graphs with mixed granularity. We consider, in following work, to use Plasma and Magma kernels to implement such graphs generated by Hydra within the StarPU runtime.

Petabricks [2] that deal with algorithmic choices in the context of high performance computing. It allows the user to specify different algorithms that can be used to solve

```
function [ A_out ] = LU_blk_var1( A, nb_alg )

  [ ATL, ATR, ...
    ABL, ABR ] = FLA_Part_2x2( A, ...
                               0, 0, 'FLA_TL');

  while ( size( ATL, 1 ) < size( A, 1 ) )

    b = min( size( ABR, 1 ), nb_alg );

    [ A00, A01, A02, ...
      A10, A11, A12, ...
      A20, A21, A22 ] = ...
      FLA_Repart_2x2_to_3x3( ATL, ATR, ...
                             ABL, ABR, b, b, 'FLA_BR');

    %----------------------------------------------%

    A01 = trilu( A00 ) \ A01;
    A10 = A10 / triu( A00 );
    A11 = A11 - A10 * A01;
    A11 = LU_unb_var1( A11 );

    %----------------------------------------------%

    [ ATL, ATR, ...
      ABL, ABR ] = ...
      FLA_Cont_with_3x3_to_2x2( A00, A01, A02, ...
                                A10, A11, A12, ...
                                A20, A21, A22, ...
                                'FLA_TL');
  end

  A_out = [ ATL, ATR
            ABL, ABR ];
  return
```

Figure 2.2: Example of Flame Input

a specific problem. The system can then explore the algorithmic choices and algorithm combinations to find the best solution for the problem on the targeted setup. It requires the user to have prior knowledge of the different algorithmic choices as well as some work to make those choices fit in the description paradigm. A description on the elemental level that will allow for composition, modularity and flexibility so that the system can define a space to explore rather than just running two single codes and compare.

Chapter 3

Hydra

Hydra is an autotuning system that, with minimal user input, will search for a parallel algorithm to solve a specified problem and produce a high performance implementation for it. Figure 3.1 gives a very high level breakdown of how Hydra works. In the simplest case, the user input is limited to the description of the problem that has to be solved and an existing kernel to solve that problem sequentially. To target heterogeneous systems, the user would also provide kernels for the different accelerators available. Theoretically, Hydra could be used on distributed systems, in which case the base kernel could be parallel to exploit resources within the nodes while Hydra would generate the inter node parallelism.

This chapter is organized as follows. Section 3.1 motivates our work through two examples. Section 3.2 presents a general overview of Hydra's components and section 3.3 describes in more details the core of the system. Finally, section 3.4 illustrates Hydra on a practical example.

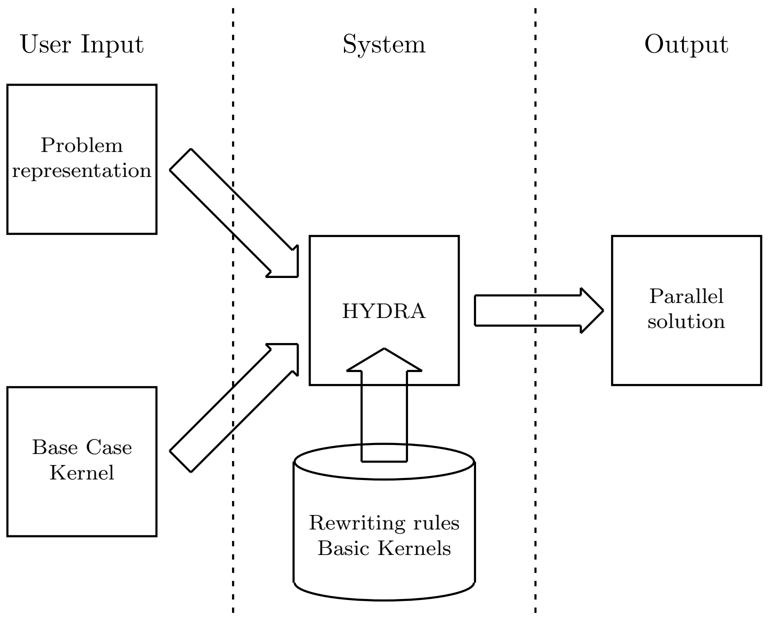


Figure 3.1: System overview

3.1 Motivating Example

In this section we will present two motivating examples. First we will talk about why we believe that parallelism should be considered from the onset of the problem and not from existing codes or algorithms. Then present the problem of tiling when seen as a data centric operation rather than a control flow operation.

3.1.1 Loss of Information

Let us consider the example of an LU decomposition. Baring properties of A, here is its formal description.

> In linear algebra, LU decomposition (also called LU factorization) is a matrix decomposition which writes a matrix as the product of a lower triangular matrix and an upper triangular matrix.

Let us now consider the Doolittle algorithm to solve LU.

Given and N x N matrix A = $(a_{n,n})$

We define $A^{(0)} = A$ and iterate $n = 1, \ldots, N-1$ as follows

Eliminate sub-diagonal elements in n-th column of $A^{(n-1)}$ by adding to the i-th row the n-th row multiplied by $l_{i,n} = \frac{a_{i,n}^{n-1}}{a_{n,n}^{n-1}}$

For $i = 1, \ldots, N$ this can be done by multiplying $A^{(n-1)}$ to the left by

$$\begin{bmatrix} 1 & & & & & 0 \\ & \ddots & & & & \\ & & 1 & & & \\ & & l_{n_1,n} & \ddots & & \\ & & \vdots & & \ddots & \\ 0 & & l_{N,n} & & & 1 \end{bmatrix}$$

We set $A^{(n)} = L_n A^{(n-1)}$

After $N-1$ steps, all sub-diagonal elements are eliminated.

$U = A^{(N-1)}$ is upper triangular.

$A = L_1^{-1} L_1 A^{(0)} = L_1^{-1} L_2^{-1} L_2 A^{(1)} = \ldots = L_1^{-1} \ldots L_{N-1}^{-1} A^{(N-1)}$

The matrix $L = L_1^{-1} \ldots L_{N-1}^{-1}$ is provably lower triangular

Thus we obtained $A = LU$.

The very first thing done in the algorithm is the definition of an iteration space. The problem of finding L and U is expressed as the problem of eliminating the sub-diagonal elements of a matrix, column by column. And there each step in this process depends on the results of the previous one, iteratively building U and building partial results for L that has to be computed at the end.

Let's now look at a specific implementation of Doolittle algorithm in figure 3.2.

```
int Doolittle_LU(double *A, int n) {
   int i, j, k, p;
   double *p_k, *p_row, *p_col;

   for (k = 0, p_k = A; k < n; p_k += n, k++) {
      for (j = k; j < n; j++) {
         for (p = 0, p_col = A; p < k; p_col += n,  p++)
            *(p_k + j) -= *(p_k + p) * *(p_col + j);
      }
      for (i = k+1, p_row = p_k + n; i < n; p_row += n, i++) {
         for (p = 0, p_col = A; p < k; p_col += n, p++) {
            *(p_row + k) -= *(p_row + p) * *(p_col + k);
         }
         *(p_row + k) /= *(p_k + k);
      }
   }
}
```

Figure 3.2: Doolittle C Implementation

The first thing that can be noticed is that the C function only has one array argument. In this implementation, the choice was made to have the output L and U be written over the input A. Thus, there is a dependence that must be examined since the memory space is shared. Secondly, due to C semantics, matrices became one dimensional arrays. Losing the information on the shape of the matrix, the number of elements is now the only piece of information available. Another implementation choice that was made here is to use pointer arithmetics rather than indexing, a compiler would now have to deal with the issue of pointer aliasing.

We can see that at the different stages of translation from the problem that is being solved to the source code produced, some information is lost, and some information appears. The devising of an algorithm not designed for parallelism can introduce sequentiality from its inception. And when implementing an algorithm the semantics of the language limit what information can be kept (e.g. matrices are linearized) and implementation choices (e.g. use of pointers within arrays rather than indexing or memory optimization) introduce new dependences. That information can limit the parallelism that can be found or make the task of exposing it more difficult. Those are the reasons why we decide in this project to start from a high level description of the problems. Trying to keep as much information as possible for as long as possible and perform our decompositions and analyses before code generation and the choice of language, frameworks and date structures.

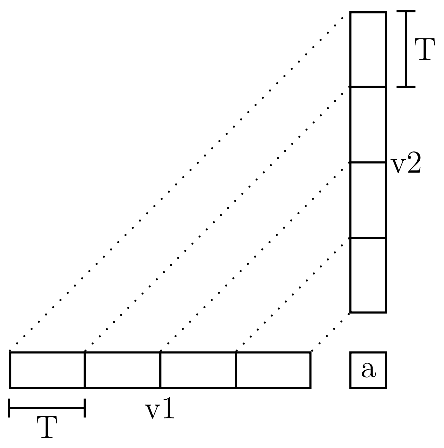
3.1.2 Tiling

Loop tiling is a well-established code transformation for program optimization for both parallelism [23] and locality [27], as well as in language extensions [7].

Tiling is typically applied on the control flow and tiles are only apparent on the data itself in the order in which it is accessed. The conceptual tiles applied to the operands are automatically set and linked by the loop nests. Figure 3.3 illustrates this for the case of the dot product. Figure 3.3a, on the left, shows the tiled code. Tiling is here parameterized by a single value, the size of the tile. On the right, figure 3.3b we illustrate what that code transformation means for the vectors involved. We can see that conceptually both vectors need to be tiled and that the number and size of those tiles must match.

```
for(int i = 0 ; i < N ; i+=T) {
  for(int it = i ; it < i + T ; it++) {
    a+= v1[it] * v2[it];
  }
}
```

(a) Source Code



(b) Vectors Visualization

Figure 3.3: Tiled Dot Product

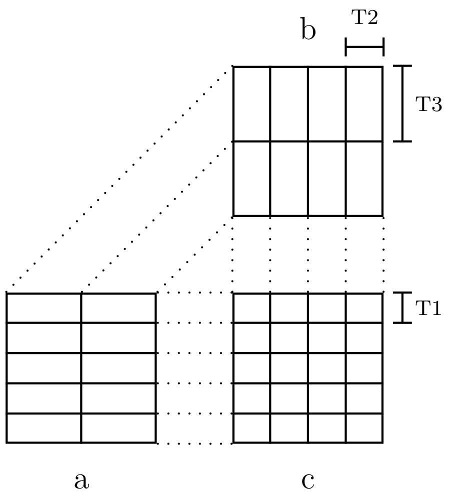
For a dot product the task seems simple. To perform tiling while focusing on the data rather than the control flow is just a matter of picking one value for a parameter shared by both vectors. However, in the general case, when considering deeper loop nests and operations on matrices, the problem becomes more complex. Figure 3.4 shows again tiling at the source level and visualized on the matrices. We can see that tiling of the operands of matrix multiplication requires six parameters (two per matrix) that are effectively paired.

```
for(int i = 0 ; i < N ; i+=T1) {
  for(int j = 0 ; j < N ; i+=T2) {
    for(int k = 0 ; k < N ; i+=T3) {

for(int it = i ; it < i+T1 ; it++) {
  for(int jt = j ; jt < j+T2 ; jt++) {
    for(int kt = k ; kt < k+T3 ; kt++) {
      c[it][ij] += a[it][kt] * b[kt][jt];
    }
  }
}

    }
  }
}
```

(a) Source Code



(b) Matrices Visualization

Figure 3.4: Tiled Matrix Multiplication

Hydra's algorithm derivation is based on tiling linear algebra equations. Operating at a high level description, it does so without control flow information, applying tiling directly on the operands. This task is achieved automatically within the framework.

3.2 General Overview

At the outer level, Hydra is built as expected for an autotuning system (Figure 3.5). A generator reads the problem description or template and generates different versions. Those versions then have to be evaluated and ranked, the objective being to decide on which version is the best, and in some cases, stop the search when the optimal is found. The latter

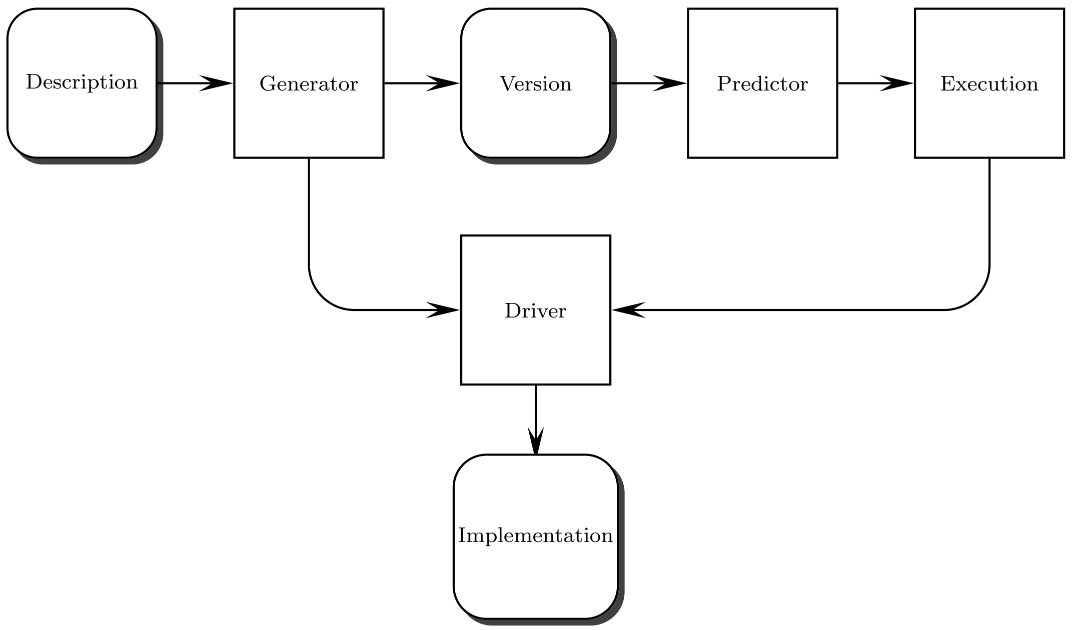


Figure 3.5: System graph

is only possible if there is a known bound to the performance achievable by the problem and/or possible on the targeted architecture. This evaluation can be achieved in two ways. First through the use of models that predict the performance by examining the algorithm or implementation. Second, through empirical benchmarking, compiling, executing and observing the performance of the versions directly. The latter is more time consuming, especially since any search space will include both good and bad versions, and the bad versions can be exponentially slower than the good ones. The former is more desirable, but very few systems can be modeled accurately and theoretical models can be quite unreliable. However, if models can't be trusted to identify optimally performing solutions, they can usually be trusted to identify codes with catastrophic performance. An autotuning system can thus be built with a two-step evaluation, using a model based predictor to filter out the bad and relying on actual execution to sort the others. The rest of this section describes the different components and their role in more details.

3.2.1 Description Language

```
%% Operands
X: Unknown Square Matrix
A: Upper Triangular Square Matrix
B: Lower Triangular Square Matrix
C: Local Unknown Square Matrix
T: Square Matrix
D: Square Matrix

%% Equation
C = T * D
A * X - X * B = C

%% Parameters
@name ctsy
@operands A B C X size
@kernel __seq_ctsy
@codelet ctsy_cl
```

Figure 3.6: Continuous Triangular Sylvester Equation Description

First, we need to provide the user with a way to describe the problem they want to solve as well as the key characteristics that we need to be able to process it. This is achieved through our description language.

Figure 3.6 shows the Continuous Triangular Sylvester Equation (CTSY) as it is described with our language. The first part of the script characterizes the operands. In this case, A and B as known Triangular Square Matrices, C as a Square Matrix and X as an Unknown Square Matrix. The keyword `Unknown` is important, as it defines the output of the program to be generated. Every operand not qualified as Unknown is considered as an input to the program. Shapes that are supported are `(Upper|Lower) Triangular` and `Symmetric`. Types are Matrices and Vectors.

The second section of the script is the equation itself. This part is very straightforward. Hydra, in its current form is limited to problems that can be expressed directly as a series of products and additions of matrices without indexing or use of specific elements. This is not an algorithmic description. The character ' can be used to specify the transpose of a matrix. The equation can be a single line or a set of equations for which dependencies are trivially determined by order of appearance of the involve operands, which is only possible since indexing cannot be used to define the equations. This notation is totally independent of the size of the problem since there is no notion of indexing but only whole matrices. Size only becomes important when generating code, although size ranges can be specified in the final section to define the search space.

The final section of the script contains customization parameters for the wanted output. All parameters have default values native to the system, but the user may specify the name desired for the generated function as well as the order of the operands (by convention, we assume a size parameter of type int is always present and named size) and the name of sequential kernel to use when generating sequential code, or the name of the codelet when generating parallel code.

Problems that we looked at are the LU decomposition (although only when it is possible without pivoting since pivoting is a problem that Hydra cannot handle natively), the triangular Sylvester equations, triangular system solvers, Cholesky and matrix multiplications and addition.

3.2.2 Generator

The core of the system is the generator. Through a set of rewriting rules it applies divide and conquer and forward substitution strategies to generate solutions to the problem. The output of the generator is a parallel task graph as well as its implementation in the selected back-end. The generator generates one version at a time, based on input parameters fed by the driver. It is described and discussed in depth in section 3.3.

3.2.3 Predictor

One of the big challenges in autotuning is to lower how long it takes to evaluate all the versions in the search space. One of the solutions to this problem is to filter ”bad” solutions and not need to actually execute them since they are the most time-consuming. Whether the objective is to examine the full space or to examine as many versions as possible in a given amount of time, faster exploration is better.

On modern architectures, performance prediction is a difficult problem. However, Hydra generated task graphs can be analyzed to provide bounds on the achievable performance for a given algorithm and decide whether to move on to actual execution or not.

The maximum breadth of the graph is the first metric that is examined. It is measured by doing a breadth first search of the graph and recording the size of the stack at every step (not counting nodes depending on unmarked predecessors). We define those values as breadth of the graph at different points. The maximum breadth gives us an approximation of the maximum number of tasks that can be running at the same time. Theoretically, the maximum cut on the graph is the actual maximum, but computing the maximum cut of a graph is a NP-hard problem, and the breadth as defined here represents a more likely occurrence in task scheduling. However, there is still no guaranty that the concerned tasks actually do end up executing in parallel (or be candidates for execution at the same time), but it does provide a fair bound on the parallelism of the algorithm. Similarly, the average breadth can be computed. For example, a graph with a maximum breadth of 2 and an average breadth of 0.5 is unlikely to perform well on a 32 core machine and can thus be discarded.

A more traditional metric, the critical path length is the second metric that can be looked at. Combined with micro benchmarking of the kernels used to build the graph, the critical path can be used to estimate a fastest termination time disregarding any overhead from scheduling or date migration. For a given graph, if the estimated fastest execution time is higher than that of a graph already measured, execution is unnecessary.

Machine learning techniques can be used by the driver to provide empirical prediction tables [25]. Every prediction is sent to the Driver.

3.2.4 Execution

The execution stage is very straightforward. The generated code for the graphs that were not filtered out by the predictor are compiled, executed and their execution is evaluated. The evaluation can be measurement of time, memory footprint, power consumption, etc. The evaluation is done on multiple runs to account for possible perturbations of the execution.

When input characteristics have an impact on the performance, the same version is evaluated on different sets of input data. In the area of dense linear algebra we are targeting, the size of the input is usually the only factor that impacts performance and it happens that for different sizes, different versions will be selected as best. Depending on code size constraints and the range of input sizes, we can generate a multi version solution that tests the input then selects the best execution path.

Sample data or data generators must be provided by the users. Although in our case, the system could be built with its own data generators.

3.2.5 Driver

The Driver is the component that directs the search. It defines the search space and prompts the generator for versions to evaluate. In its simplest version, the driver is a simple script performing exhaustive search of the defined space, and returning the best solution from the execution measurements. It relies solely on the predictor to constrain search time.

There exist many machine learning search strategies that could be implemented in the driver to further improve the overall system. This is, however, a purely engineering matter and not the focus of our research and we did not investigate further than exhaustive search.

3.3 In Depth View of the Generator

At the core of Hydra, the generator is the main contribution described in this thesis. It is the component that will analyze the problem description fed to the system, analyze it, and determine how to break it down into a parallel task graph. It has the charge of translating mathematical equations into tasks and building the dependence graph connecting those tasks.

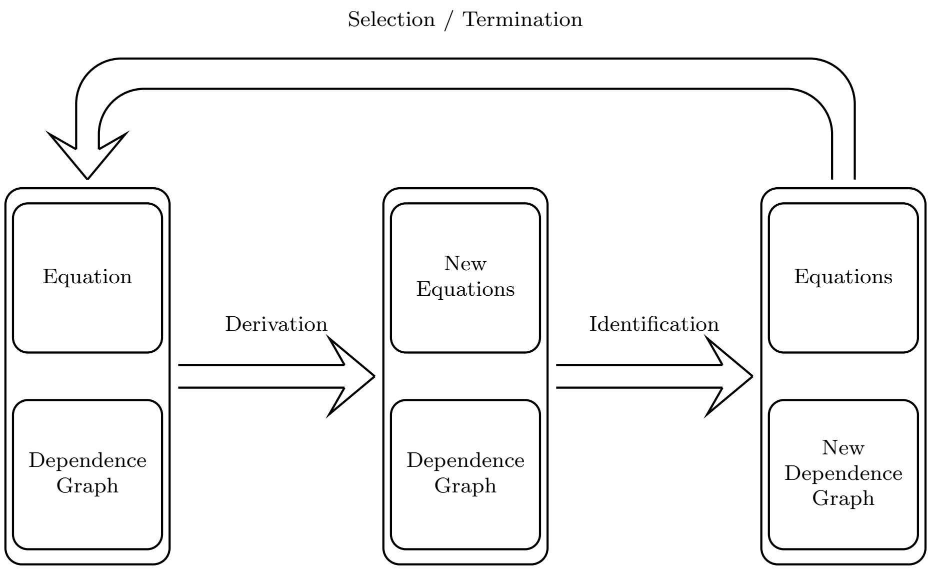


Figure 3.7: Generator overview

The generator operates on a tuple composed of a set of equations and a directed graph whose vertices are tasks and edges represent dependencies. The edges are labeled with the operand that carry the dependence.

Figure 3.7 illustrates the main steps of the generation process. The generator operates on a single equation at a time. Generation can be decomposed into two major steps: derivation (section 3.3.1) that will decompose the equation through tiling of its operands into smaller ones, and identification (section 3.3.2) that maps equations to tasks, finds the dependencies between the new tasks, then merges the localized dependence graph with the general one.

This process can be applied recursively, further tiling generate equations to generate smaller granularity paths in the task graph. Initially, the set contains the single equation provided by the user and the dependence graph is a single vertex with no edge.

It is interesting to note that when a homogeneous architecture is targeted, recursion is unnecessary. Indeed, tiling one level by x and the next by y leads to the same graph as tiling by $x * y$ at the top level. The interest of recursive tiling is to generate graphs of uneven granularity to bias execution paths towards different processing units on heterogeneous systems.

The generation process terminates when the characteristics of the graph passed as arguments are met. Those parameters are tiling and recursion factors.

Example

Consider the equation $M = L \cdot X$ with M a known matrix, L a known lower triangular matrix and X the unknown matrix. One way to derive this equation is to tile its operands twice in each dimension. How valid derivations are discovered is discussed in section 3.3.1. After tiling, the equation can be written with each operand as a matrix of smaller matrices:

$$\begin{pmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{pmatrix} \cdot \begin{pmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{pmatrix}$$

Through symbolic execution, we can compose the equation into the following set:

$$\begin{cases} M_{00} = L_{00} \cdot X_{00} & (3.1a) \\ M_{01} = L_{00} \cdot X_{01} & (3.1b) \\ M_{10} = L_{10} \cdot X_{00} + L_{11} \cdot X_{10} & (3.1c) \\ M_{11} = L_{10} \cdot X_{01} + L_{11} \cdot X_{11} & (3.1d) \end{cases}$$

The dependence graph associated with the initial $M = L \cdot X$ equation is a single node. Identification leads to equations 3.1a and 3.1b being matched to the original problem and are thus being marked as tasks producing their outputs X_{00} and X_{01} respectively. Matching is done by comparing the signature built from the sequence of operations and the operand shapes, the process is described in details in section 3.3.2. Dependence edges are created between them and other unidentified tasks that use those operands. Equations 3.1c and 3.1d cannot be identified as they are and would be further decomposed into three smaller equations matching available kernels.

3.3.1 Derivation (Equation Expansion through Tiling)

Equation derivation is the first step to generating a parallel algorithm for a given problem. It looks at a single equation with information on its operands and generates a set of new equations. This stage does not consider semantic meaning of the generated equations, it simply applies tiling to the input equation's operands and performs symbolic execution. Turning those equations into tasks and finding the dependences between them are done later on in the Identification step.

Validity of Tiling

Some basic rules have to hold for a tiling to be valid for the derivation. Although in a full framework run, the driver has the charge of the validity of the search space, the generator contains logic to verify that it generates only valid solutions. Tiling is applied to the matrix operands with the objective to decompose the problem into a set of valid equations through symbolic execution. Thus the tiling must conform to the rules of basic matrix operations. To multiply two matrices A and B, the number of columns in A must be equal to the number of rows in B. To add two matrices A and B, the number of rows and columns of A and B must be identical. The same constraints must hold in the number of blocks in each operand's dimension according to the operations performed in the equation. Any tiling that does not conserve those rules is invalid, and should not be considered as part of the search space.

The definition of the search space is achieved by linking the operands' dimensions through the use of the operation tree and propagation of matrix properties. It is assumed that the matrices themselves are of appropriate sizes for all operations part of the equation and thus do not worry about them, but only about the number of blocks.

For the purpose of identification, some extra constraints may be "artificially" added to ensure the transfer of shapes. For example, when tiling a triangular or diagonal matrix, we would enforce square tiles to ensure that all diagonal blocks are themselves triangular or diagonal.

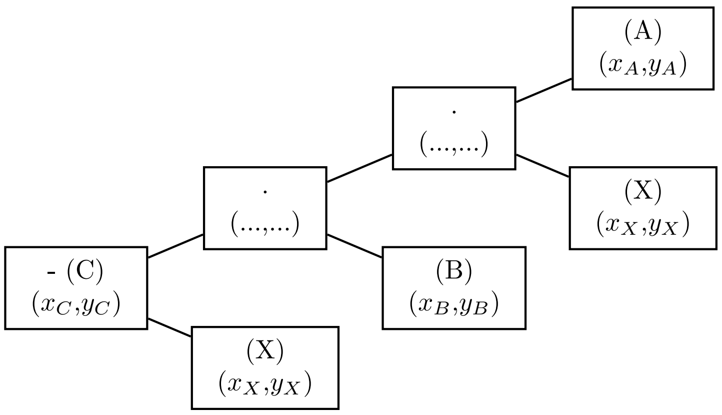


Figure 3.8: Operation Tree

Figures 3.8 through 3.10 illustrate the first step of the process. Where we propagate the operands' dimensions. For this example, we consider the equation $A \cdot X \cdot B - X = C$.

First (Figure 3.8) the system creates the operation tree assigning a tuple (x,y) to each

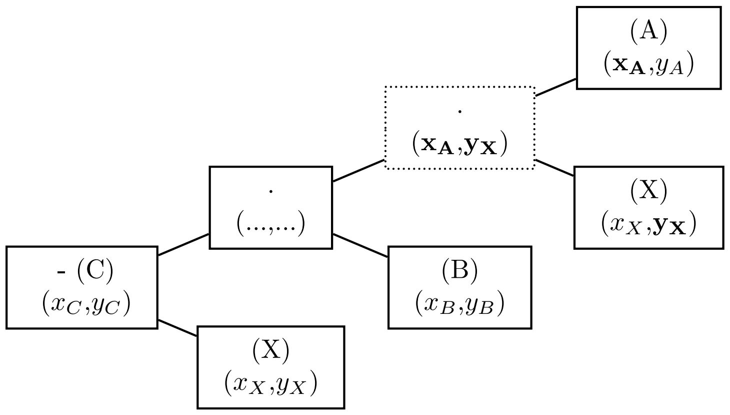


Figure 3.9: First Dimension Propagation

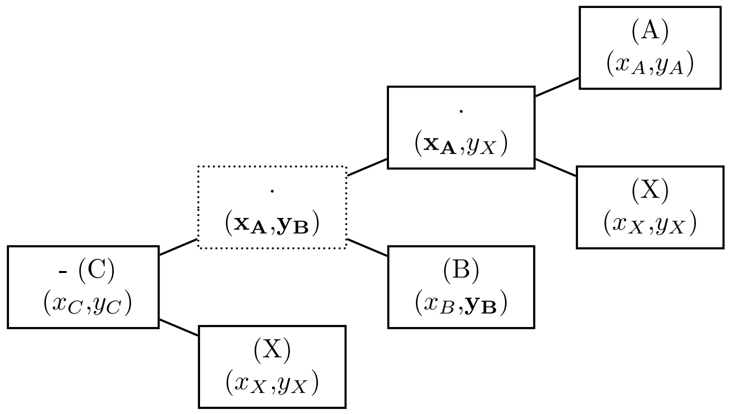


Figure 3.10: Second Dimension Propagation

operand where x and y are the number of tiles per column and row respectively. Real operands correspond to the leaves and the root of the tree, they are named in the original equation. Virtual operands are inner nodes of the tree, they correspond to intermediate computation. For example, computing $A \cdot X \cdot B$ can be decomposed in two steps, where we first compute $T = A \cdot X$ then $T \cdot B$ the dimensions of T are determined from that of A and X and in this case are used to link the dimensions of B to that of the other operands in the equation.

Then, starting at the leaves, we go through the operation tree to assign the dimensions to all the inner nodes of the operation tree. Those are deduced from the dimensions of the operands and the operation itself. The result of a matrix addition is of the same dimensions as both operands. The product of an $m\times n$-matrix by an $n\times l$-matrix will produce an $m\times l$-matrix. For example, in figure 3.9, the tuple (x_A, y_X) to the result node of $A \cdot X$ and, in figure 3.10, (x_A, y_B) for result node of $A \cdot X \cdot B$

Now that all the nodes and leaves of the operation tree are properly populated, the system examines all the operations in the tree to generate the constraints that must hold on each operand dimension. For example, looking the product $A \cdot X$, the constraint $y_A = x_X$ is generated. The system 3.2 is composed of all the constraints generated by this process.

$$\begin{cases} y_A = x_X \\ y_X = x_B \\ x_A = x_X \\ y_B = y_X \\ x_C = x_X \\ y_C = y_X \end{cases} \tag{3.2}$$

This system can be consolidated using algorithm 1 to produce sets of variables that must be equal the number of such sets defines the number of dimensions of our search space at the top level. Equations 3.3 illustrate this with our example, we have here a two dimensional search space.

Algorithm 1 Constraint Consolidation

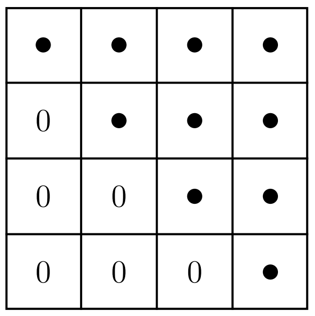
```
Require: Set $C$ of constraints
Require: Empty Set $O$ containing output
1: for all $(c_1, c_2) \in C$ do
2:    $found \leftarrow$ false
3:    for all $o \in O$ do
4:       if $c_1 \in o$ and $c_2 \notin o$ then
5:          $o \leftarrow o \cup \{c_2\}$
6:          $found \leftarrow$ true
7:       if $c_2 \in o$ and $c_1 \notin o$ then
8:          $o \leftarrow o \cup \{c_1\}$
9:          $found \leftarrow$ true
10:   if not found then
11:      $O \leftarrow O \cup \{\{c_1, c_2\}\}$
```

$$\begin{aligned} \{x_A = y_A &= x_X = x_C\} \\ \{x_B = y_B &= y_X = y_C\} \end{aligned} \tag{3.3}$$

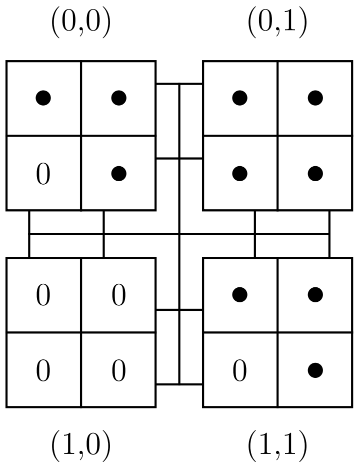
Tiling

Hydra discovers algorithms of Divide and Conquer nature. The division is achieved through tiling of the operands followed by a symbolic execution of the equation, considering the operands as matrices of tiles. The result of symbolic execution is a list of equations on the tiles of the original operands.

A couple of other properties of linear algebra are exploited by Hydra to achieve its goal. Namely, those of 0-tiles (A matrix tile that contains no non zero element). The 0-matrix is the absorbing element for matrix multiplication, pre or post multiplying any matrix by the 0-matrix produces the 0-matrix ($\forall$ n-by-m matrix X, $0 \cdot X = 0$). The 0-matrix is the identity element for matrix addition, adding any matrix the 0-matrix produces that same matrix ($\forall$ n-by-m matrix X, $0 + X = X$). Using the shapes of the operands, we can identify tiles that are 0-matrices and use that information to identify superfluous computation and simplify the generated equations.



(a) Original Upper Triangular Matrix



(b) Tiled Upper Triangular Matrix

Figure 3.11: Shape Aware Matrix Tiling

Figure 3.11 illustrates how shapes are used when tiling an upper triangular matrix (3.11a). For the example a 2-by-2 tiling is applied. Figure 3.11b shows the different tiles, we can see that blocks `(0,0)` and `(1,1)` are upper triangular and block `(1,0)` is a 0-matrix. The latter will be used immediately in the derivation stage, the former stored for later use in identification.

$$T = A \cdot X \tag{3.4}$$

Let us consider equation (3.4) where A is a square upper triangular matrix and X and T are square matrices, and apply 2-by-2 tiling to its operand as illustrated in (3.5)

$$\begin{bmatrix} T_{00} & T_{01} \\ T_{10} & T_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{bmatrix} \tag{3.5}$$

Full symbolic execution of (3.5) leads to the system of equations in (3.6)

$$\begin{cases} T_{00} = A_{00} \cdot X_{00} + A_{01} \cdot X_{10} \\ T_{01} = A_{00} \cdot X_{01} + A_{01} \cdot X_{11} \\ T_{10} = A_{10} \cdot X_{00} + A_{11} \cdot X_{10} \\ T_{11} = A_{10} \cdot X_{01} + A_{11} \cdot X_{11} \end{cases} \tag{3.6}$$

As illustrated in figure 3.11b A_{10} is a 0-matrix and equations containing it can be simplified. A naive way to deal with 0-blocks is to perform full symbolic execution (leading to (3.6)) then identifying 0-matrices and simplifying the system to produce (3.7). However, the location of 0-tiles is known from the shapes of the matrices, and it is a trivial matter, through inequalities, to ignore 0-tiles during symbolic execution and produce directly the desired set of equations.

$$\begin{cases} T_{00} = A_{00} \cdot X_{00} + & A_{0,1} \cdot X_{10} \\ T_{01} = A_{00} \cdot X_{01} + & A_{0,1} \cdot X_{11} \\ T_{10} = & A_{11} \cdot X_{10} \\ T_{11} = & A_{11} \cdot X_{11} \end{cases} \tag{3.7}$$

3.3.2 Identification and Dependence Graph Computation

Derivation of the original problem decomposed the problem into a set of new equations. However, at this stage, those equations are just abstract views. To actually build a solution to the problem, we need to determine how to solve each new equation as well as the order in which to solve them. The next step is to identify the equations by linking them to tasks (i.e. available kernels) and building the dependence graph that will guide their execution.

Signatures and Simplification

Equation signatures are the cornerstone of the identification process. The signature of an equation is built as a combination of its operations and the shape of its operands.

Let the following abbreviations stand to represent signatures textually. The actual representation used inside the compiler is a purely engineering matter that does not impact the theory presented here.

- `LT` : Known Lower Triangular Matrix
- `UT` : Known Upper Triangular Matrix
- `MT` : Known Matrix of Unspecified shape
- `UNK` : Unknown matrix
- `UNK_LT` : Unknown Lower Triangular matrix
- `UNK_UT` : Unknown Upper Triangular matrix
- `*` : Matrix multiplication
- `+` : Matrix Addition
- `-` : Matrix Subtraction
- `=` : Equality

For example, $T = A \cdot X$ with A upper triangular, would have the following signature: `MT = UT * UNK`

Within this typing system, we state that two equations that have the same signature are, in fact, the same equation. We define the equal sign in an equation as symmetrical when comparing two signatures. `MT = UT * UNK` and `UT * UNK = MT` would thus be identified as identical signatures. The system builds a base of signatures corresponding to problems it has a kernel for. In particular, it allows to identify instances of the original problem.

Equations are not always directly identifiable. Some simplification rules are defined on signatures that allow to build the canonical signature of an equation. This allows the identification of equations that can be made into matches to known problem through expansion. Expansion will be explained and discussed later on in this section.

A few examples of such simplification rules are:

- `MT + MT` $\Rightarrow$ `MT`
- `MT - MT` $\Rightarrow$ `MT`
- `MT * MT` $\Rightarrow$ `MT`
- `... + MT = MT` $\Rightarrow$ `... = MT - MT`

Every rule is expressed for every combination of type, since it happens that shapes can be preserved by the operations depicted.

- `LT * LT` $\Rightarrow$ `MT`
- `LT + LT` $\Rightarrow$ `LT`

Simplification steps are applied iteratively until no rule is applicable and the result is considered the canonical signature for the equation considered.

Finally, relaxation rule are defined on signatures. Some types are weaker than others and can be used as alternatives.

- `LT` $\Rightarrow$ `MT`
- `UT` $\Rightarrow$ `MT`

It may be the case that for a particular equation, after derivation, matrix multiplications are performed on triangular matrices. But that no kernel is defined to deal specifically with that case, whereas a kernel is known for matrix multiplication in general. In such a case, identification would require the extra relaxation step to proceed.

Example

To illustrate equation simplification, let us examine equation (3.8) derived from $L \cdot X + X \cdot U = M$ with M a known matrix, L a known lower triangular matrix, U a known upper triangular matrix and X and unknown matrix.

$$L_{00} \cdot X_{01} + X_{00} \cdot U_{01} + X_{01} \cdot U_{11} = M_{01} \tag{3.8}$$

The signature of the original problem is `LT * UNK + UNK * UT = MT` and considering X_{00} to be known, the current equation's signature is `LT * UNK + MT * MT + UNK * UT = MT` and does not match the original problem nor any of the basic built in problems.

```
     LT * UNK + MT * MT + UNK * UT = MT
⇔         LT * UNK + MT + UNK * UT = MT
⇔         LT * UNK + UNK * UT = MT - MT
⇔              LT * UNK + UNK * UT = MT
```

After building the canonical signature however, it is found that (3.8) is an instance of the original problem. The steps required to make this apparent on the actual equation is called expansion and is described later on in this section.

Identification

Algorithm 2 Equation Selection

```
Require: Set $E$ of equations
 1: for all $e \in E$ do
 2:    if $e.signature = main.signature$ then
 3:       return e
 4: for all $e \in E$ do
 5:    if $|e.output| = |main.output|$ then
 6:       if simplification(e.signature,main.signature) then
 7:          $e \leftarrow expand(e)$
 8:          return e
 9: for all $e \in E$ do
10:    if $|e.output| = 1$ and $solvable(e)$ then
11:       return e
12: print Error
```

Identification is the process of finding an equation that can be matched to one or multiple kernels and thus made into a task or task graph. Identification operates on a set of unidentified equations and returns an identified equation. Algorithm 2 details how this is achieved.

Due to the nature of our approach, we expect to find instances of the original problem in the set of newly generated equations. This is the first thing that we look for.

First (line 1), we look for direct matches, equations that are exact instances of the original, on tiles of the original operands. Such equations have the same signature without any need for simplification.

Then (line 4), we look for equations whose canonical signature are a match to that of the original problem's. Such equations are intuitively, equations that can be decomposed into a string of dependent tasks with the last task being an instance of the original problem. For example, equation $L \cdot X + B = M$ with L lower triangular and X unknown directly solve by a kernel for problems matching signature `LT * UNK = M`. However, its canonical signature is a match, if expanded (line 7 into $T = M - B$ and $L \cdot X = R$ with a dependence carried by T between the two equations, we find an instance of a basic operation (matrix subtraction) and a direct match to the original problem. In this case, the output of identification is a small dependence graph rather than a single equation.

Finally, if no equation is found that matches the original or can be made to match the original, we look for simple equations that produce a single output and are directly solvable (e.g. a matrix multiplication of the form Unknown = Known * Known)

When identification fails, equations that failed to be identified are displayed to the user along with their signature and canonical signature. The user then has the option of providing a kernel to solve such equations, and on a new run, identification would work.

Expansion

Expansion is the process by which an equation that was identified through signature simplification is translated into a set of known tasks. The process through which this is achieved is rather straight forward. The simplification rules applied to a signature to reduce it to canonical form are all essentially the reduction of a matrix operation to its result. We can thus follow the simplification steps, mapping the signature identities to the actual operands. And then, for every signature operation that is collapsed, we can introduce a temporary as the result of the corresponding operation and rewrite the original equation using that temporary. The basic matrix operations that are used for matrix operations have native kernels associated in Hydra. Thus every equation introduced in expansion is already identified and associated with an existing kernel. The dependences between the generated equations are trivial and immediate.

Example

The following table illustrates expansion on a problem. The left column shows the steps taken to simplify a signature into canonical form, and the right column the associated equations created.

	`LT * UNK + MT * MT + UNK * UT = MT`	$L_{00} \cdot X_{01} + X_{00} \cdot U_{01} + X_{01} \cdot U_{11} = M_{01}$
$\Leftrightarrow$	`LT * UNK + MT + UNK * UT = MT`	$T_0 = X_{00} \cdot U_{01}$ $L_{00} \cdot X_{01} + T_0 + X_{01} \cdot U_{11} = M_{01}$
$\Leftrightarrow$	`LT * UNK + UNK * UT = MT - MT`	
$\Leftrightarrow$	`LT * UNK + UNK * UT = MT`	$T_0 = X_{00} \cdot U_{01}$ $T_1 = M_{01} - T_0$ $L_{00} \cdot X_{01} + X_{01} \cdot U_{11} = T_1$

The introduction of temporaries can, in practice, lead to high memory footprint and have a negative impact on performance. This can be counteracted in two different manners. First in the code generator, exploiting properties and functionalities of the back end (see section 4.1 for the choices we made evaluating Hydra). Another solution, is to look at this as a register allocation problem, since intermediate matrices are usually short lived, it is possible to reuse memory along execution paths. However, we observed empirically that the minimum number of allocated memory can create extra memory transfer and harm performance. The choice of how much reuse to add for intermediate matrices is not trivial. This is discussed in section 4.1.

Building Dependences

With the different tools described to this point, we can now work on building the dependence graph that will represent a full parallel solution. The first step is to determine input and output sets for all the equations created from derivation. As discussed in section 3.2.1, output variable are defined by the user as `Unknown` while every other variable is known, i.e.

an input. When tiling is performed, the information on the original operands is kept. And since tiling does not change the content of a matrix, tiles of known matrices are considered to be known, and tiles of unknown matrices are considered to be unknown.

For every equation, we thus built initial input and output sets, establishing the starting point of the identification process.

Example

Let us consider equation $T = A \cdot X$. In this case, A and T are known matrices, so after two by two tiling of all operands in derivation, we have L_{ij} and M_{ij} for $i, j \in \{0, 1\}$ also marked as known. Similarly, since matrix X is unknown, matrices X_{ij} for $i, j \in \{0, 1\}$ are unknown.

With that information established, equations are now identifiable and we can start the process (Algorithm 3). The first step (line 2) is to identify an equation in the set of unidentified derivated equations. Once an equation is identified, it is removed from the set and placed in the dependence graph. In the case where e is a set of equations (identification through canonical signature and expansion) the previously inbound dependence arcs must be connected to the proper equations that now compose e.

Algorithm 3 Building the Dependence Tree

```
1:  while $E \neq \emptyset$ do
2:    Identify $e$ in $E$
3:    $E \leftarrow E \setminus \{e\}$
4:    Merge $D$ and $e$
5:    for all $o \in e.output$ do
6:      for all $d \in E$ do
7:        if $o \in d.output$ then
8:          $D \leftarrow D \cup \{(e \rightarrow d[o])\}$
9:          $d.output \leftarrow d.output \setminus \{o\}$
10:         $d.input \leftarrow d.input \cup \{o\}$
```

Once an equation is identified, it is "executable" at this stage in the task graph (i.e.

enough information is available that it could be scheduled for execution), its input set is available and it produces values for all matrices in its output set. The next step is to reflect that fact on the remaining equations in E (line 5): iterating over all equations in E, we examine the content of the output sets, looking for instances of each output of the newly identified equation. Upon finding such a matrix, we can remove it from the output set (it is no longer an unknown entity) and add it to the input set. There is now a data dependence between those two equations (line 8) carried by the operand (o). In the algorithm, it is denoted by an arrow from the equation generating the matrix to the equation using it, tagged with the matrix itself.

Once E is empty, every equation has been identified and added to the dependence graph. At which stage the only task left to produce a complete task graph solving the original equation is to add the extra tasks necessary to reshape the data. Hydra operates purely logical representations of the problem during the generation process. However, a code generator would only have as available information, the original operands and it is thus necessary to define explicitly what the different A_{ij} translate to. A task graph is considered complete once, for every node in the graph, there is one incoming dependence arc for each of the matrices in the input set. Intermediate matrices introduced in expansion already satisfy this by construction. What is left are the tiles of input variables. Extra tasks are created to create the tiles from the original matrices. Logically, such tasks are called copies, have no inputs (thus satisfying the condition for graph completeness) and a single output.

Similarly, the task graph generated by Hydra, at this stage, does not produce the Unknown required by the user, but rather, pieces of it in different tasks. It is thus necessary to add "copy out" tasks that copy the unknown tiles back into the original unknown to return the result desired by the user.

Example

Let us consider once more the equation $T = A \cdot X$ where T is a known matrix, A is a known upper triangular matrix and X is an unknown matrix. Two by two tiling is applied to each of the operands. The starting point for full identification is shown in the following table.

	Equation	Input set	Output set
(1)	$T_{00} = A_{00} \cdot X_{00} + A_{01} \cdot X_{10}$	$\{T_{00}, A_{00}, A_{01}\}$	$\{X_{00}, X_{10}\}$
(2)	$T_{01} = A_{00} \cdot X_{01} + A_{01} \cdot X_{11}$	$\{T_{01}, A_{00}, A_{01}\}$	$\{X_{01}, X_{11}\}$
(3)	$T_{10} = A_{11} \cdot X_{10}$	$\{T_{10}, A_{11}\}$	$\{X_{10}\}$
(4)	$T_{11} = A_{11} \cdot X_{11}$	$\{T_{11}, A_{11}\}$	$\{X_{11}\}$

Dependence Graph (D)
$\emptyset$

Equation (3) is a match to the original problem and is directly identified as producing X_{10}. Post identification, since (1) has X_{10} in its output set, a dependence carried by X_{10} is created between (3) and (1) ($D = D \cup (3) \rightarrow (1)[X_{10}]$). (4) is identified the same way, with a dependence carried by X_{11} to (2).

	Equation	Input set	Output set
(1)	$T_{00} = A_{00} \cdot X_{00} + A_{01} \cdot X_{10}$	$\{T_{00}, A_{00}, A_{01}, X_{10}\}$	$\{X_{00}\}$
(2)	$T_{01} = A_{00} \cdot X_{01} + A_{01} \cdot X_{11}$	$\{T_{01}, A_{00}, A_{01}, X_{11}\}$	$\{X_{01}\}$

Dependence Graph (D)
$\{(3) \rightarrow (1)[X_{10}]; (4) \rightarrow (2)[X_{11}]\}$

At this stage, both (1) and (2) are identifiable via their canonical signature. Both have the same signature: `MT = UT * UNK + MT *MT` that does not match any known problem. However, their canonical signature is a match to the original problem : `MT =`

`UT * UNK.`

Looking at equation (3), it reads T_{10} and A_{11} and produces the tile X_{10} of the original unknown. The following additional tasks are thus created with their associated dependencies:

$$\begin{array}{lll}(\text{c1}) & \text{copy\_in } T_{10} & (c1) \rightarrow (3)[T_{10}] \\ (\text{c2}) & \text{copy\_in } A_{11} & (c2) \rightarrow (3)[A_{11}] \\ (\text{c3}) & \text{copy\_out } X_{10} & (3) \rightarrow (c3)[X_{10}]\end{array}$$

3.4 Practical Example: Discrete Triangular Sylvester Equation (DTSY)

The discrete formulation of the triangular Sylvester equation is $L \cdot X \cdot U - X = C$ with L a lower triangular matrix, U an upper triangular matrix, C a square matrix and X the output, an unknown square matrix.

Tiling validity analysis produces the following sets:

$$\{x_L = y_L = x_X = x_C\}$$
$$\{x_U = y_U = y_X = y_C\}$$

In this section, we will illustrate how the generator works on a single point in the exploration space. In a full Hydra run, the process would be applied for all point within the bounds set by the user. For clarity and simplicity, we will look at the point corresponding to a two by two tiling of all operands. Derivation produces equations (3.9).

$$\begin{cases} L_{00} \cdot X_{00} \cdot U_{00} - X_{00} = C_{00} & \text{(3.9a)} \\ L_{10} \cdot X_{00} \cdot U_{00} + L_{11} \cdot X_{10} \cdot U_{00} - X_{10} = C_{10} & \text{(3.9b)} \\ L_{00} \cdot X_{00} \cdot U_{01} + L_{00} \cdot X_{01} \cdot U_{11} - X_{01} = C_{01} & \text{(3.9c)} \\ L_{10} \cdot X_{00} \cdot U_{01} + L_{11} \cdot X_{10} \cdot U_{01} + L_{10} \cdot X_{01} \cdot U_{11} + & \\ \qquad L_{11} \cdot X_{11} \cdot U_{11} - X_{11} = C_{11} & \text{(3.9d)} \end{cases}$$

As a reminder, following are the symbols use to express equation signatures:

- `LT` : Known Lower Triangular Matrix
- `UT` : Known Upper Triangular Matrix
- `MT` : Known Matrix of Unspecified shape
- `UNK` : Unknown matrix

Furthermore, subscripts are used to differentiate operands in the signature. This differentiation is especially important to expose the number of unknowns in an equation, and identify matching problems.

We first define the signature of the original equation (3.10). In the rest of this section, we will refer to it as the original signature.

$$\texttt{LT * UNK * UT - UNK = MT} \tag{3.10}$$

We then start the Identification process by building the initial input and output table. At this stage, all tiles inherit their status from the original matrix. Tiles of unknowns are outputs and tiles of knowns are inputs. We are going to represent the steps in the process in the form of tables. Each table contains an entry for each equation that still needs to be identified. The last line, separated by a double line from the upper half of the table is labeled

Dependencies and contains the dependence graph as a list of dependence edges. Equations are identified by the unique labels specified in their respective systems. An equation entry consist as its label, its input set, output set and finally signature.

The following table presents the initial state of the system

Equation	Input	Output	Signature
(3.9a)	$L_{00}; U_{00}$	X_{00}	`LT * UNK * UT - UNK = MT`
(3.9b)	$L_{10}; U_{00}$ L_{11}	$X_{00}; X_{10}$	$\texttt{MT}_1$ `*` $\texttt{UNK}_1$ `* UT + LT *` $\texttt{UNK}_2$ `* UT` `-` $\texttt{UNK}_2$ `=` $\texttt{MT}_2$
(3.9c)	$L_{00}; U_{01}$ U_{11}	$X_{00}; X_{01}$	`LT *` $\texttt{UNK}_1$ `*` $\texttt{MT}_1$ `+ LT *` $\texttt{UNK}_2$ `* UT` `-` $\texttt{UNK}_2$ `=` $\texttt{MT}_2$
(3.9d)	$L_{10}; U_{01}$ $L_{11}; U_{11}$	$X_{00}; X_{01}$ $X_{10}; X_{11}$	$\texttt{MT}_1$ `*` $\texttt{UNK}_1$ `*` $\texttt{MT}_2$ `+ LT *` $\texttt{UNK}_2$ `*` $\texttt{MT}_2$ `+` $\texttt{MT}_1$ `*` $\texttt{UNK}_3$ `* UT + LT *` $\texttt{UNK}_4$ `* UT` `-` $\texttt{UNK}_4$ `=` $\texttt{MT}_3$
Dependencies	$\emptyset$		

With its signature a direct match to the original signature, equation (3.9a) is the first to be identified. As a direct match, it is directly mapped to a task and becomes the root of the dependence graph being built. X_{00} is now a known matrix and all input and output sets for the remaining equations are updated accordingly. Dependence edges are created between (3.9a) and (3.9b), (3.9c) and (3.9d) since they all require X_{00}.

Equation	Input	Output	Signature
(3.9b)	$L_{10}; U_{00}$ $L_{11}; X_{00}$	X_{10}	$\texttt{MT}_1$ `*` $\texttt{MT}_2$ `* UT + LT * UNK * UT - UNK =` $\texttt{MT}_3$
(3.9c)	$L_{00}; U_{01}$ $U_{11}; X_{00}$	X_{01}	`LT *` $\texttt{MT}_1$ `*` $\texttt{MT}_2$ `+ LT * UNK * UT - UNK =` $\texttt{MT}_3$
(3.9d)	$L_{10}; U_{01}$ $L_{11}; U_{11}$ X_{00}	$X_{10}; X_{01}$ X_{11}	$\texttt{MT}_1$ `*` $\texttt{MT}_4$ `*` $\texttt{MT}_2$ `+ LT *` $\texttt{UNK}_1$ `*` $\texttt{MT}_2$ `+` $\texttt{MT}_1$ `*` $\texttt{UNK}_2$ `* UT + LT *` $\texttt{UNK}_3$ `* UT -` $\texttt{UNK}_3$ `=` $\texttt{MT}_3$
Dependencies	(3.9a)→(3.9b) [X_{00}] ; (3.9a)→(3.9c) [X_{00}] ; (3.9a)→(3.9d) [X_{00}]		

Equations (3.9b) and (3.9c) are both identifiable at this stage. But neither is a direct matches and their signature have to be simplified to canonical form to be successfully identified.

$$\begin{aligned}
& \texttt{MT}_1 * \texttt{MT}_2 * \texttt{UT + LT * UNK * UT - UNK} && = \texttt{MT}_3 \\
\Leftrightarrow\quad & \texttt{MT}_1 * \texttt{UT + LT * UNK * UT - UNK} && = \texttt{MT}_2 \\
\Leftrightarrow\quad & \texttt{MT}_1 \texttt{ + LT * UNK * UT - UNK} && = \texttt{MT}_2 \\
\Leftrightarrow\quad & \texttt{LT * UNK * UT - UNK} && = \texttt{MT}
\end{aligned}$$

Since the equation was identified through its canonical signature, an expansion step is necessary. System (3.11) contains the new equations that replace (3.9b).

$$(3.9\text{b}) \Rightarrow \begin{cases} T_1 = L_{10} \cdot X_{00} & (3.11\text{a}) \\ T_2 = T_1 \cdot U_{00} & (3.11\text{b}) \\ T_3 = C_{10} - T_2 & (3.11\text{c}) \\ L_{11} \cdot X_{10} \cdot U_{00} - X_{10} = T_3 & (3.11\text{d}) \end{cases}$$

Equation (3.9c) is processed the same way. With both (3.9b) and (3.9c) identified, X_{10} and X_{01} are transferred to the input set of the last equation.

Equation	Input	Output	Signature
(3.9d)	$L_{10}; U_{01}$ $L_{11}; U_{11}$ $X_{00}; X_{10}; X_{01}$	X_{11}	$\texttt{MT}_1$ `*` $\texttt{MT}_2$ `*` $\texttt{MT}_3$ `+ LT *` $\texttt{MT}_4$ `*` $\texttt{MT}_3$ `+` $\texttt{MT}_1$ `*` $\texttt{MT}_5$ `* UT + LT * UNK * UT - UNK =` $\texttt{MT}_6$

Dependencies	(3.9a) $\rightarrow$ (3.11a) $[X_{00}]$; (3.9a) $\rightarrow$ (3.9c) $[X_{00}]$; (3.9a) $\rightarrow$ (3.9d) $[X_{00}]$ (3.11a) $\rightarrow$ (3.11b) $[T_1]$; (3.11b) $\rightarrow$ (3.11c) $[T_2]$; (3.11c) $\rightarrow$ (3.11d) $[T_3]$ (3.11c) $\rightarrow$ (3.9d) $[X_{10}]$; (3.9c) $\rightarrow$ (3.9d) $[X_{10}]$;

At this stage (3.9d) is identified and a full dependence graph has been built. System (3.12) contains all the equations identified, for clarity and space, basic matrix operations were clustered together. For example, equations (3.11a), (3.11b) and (3.11c) are combined into (3.12b). Figure 3.12 shows the final task graph, the copy tasks were excluded for clarity. The nodes are labeled with the output they produce.

$$\begin{cases}
L_{00} \cdot X_{00} \cdot U_{00} = C_{00} \rightarrow [X_{00}] & \text{(3.12a)} \\
C_{10} - L_{10} \cdot X_{00} \cdot U_{00} = T_1 \rightarrow [T_1] & \text{(3.12b)} \\
L_{11} \cdot X_{10} \cdot U_{00} - X_{10} = T_1 \rightarrow [X_{10}] & \text{(3.12c)} \\
C_{01} - L_{00} \cdot X_{00} \cdot U_{01} = T_2 \rightarrow [T_2] & \text{(3.12d)} \\
L_{00} \cdot X_{01} \cdot U_{11} - X_{01} = T_2 \rightarrow [X_{01}] & \text{(3.12e)} \\
L_{10} \cdot X_{00} \cdot U_{01} = T_3 \rightarrow [T_3] & \text{(3.12f)} \\
L_{11} \cdot X_{10} \cdot U_{01} = T_4 \rightarrow [T_4] & \text{(3.12g)} \\
L_{10} \cdot X_{01} \cdot U_{11} = T_5 \rightarrow [T_5] & \text{(3.12h)} \\
C_{11} - T_3 - T_4 - T_5 = T_6 \rightarrow [T_6] & \text{(3.12i)} \\
L_{11} \cdot X_{11} \cdot U_{11} - X_{11} = T_6 \rightarrow [X_{11}] & \text{(3.12j)}
\end{cases}$$

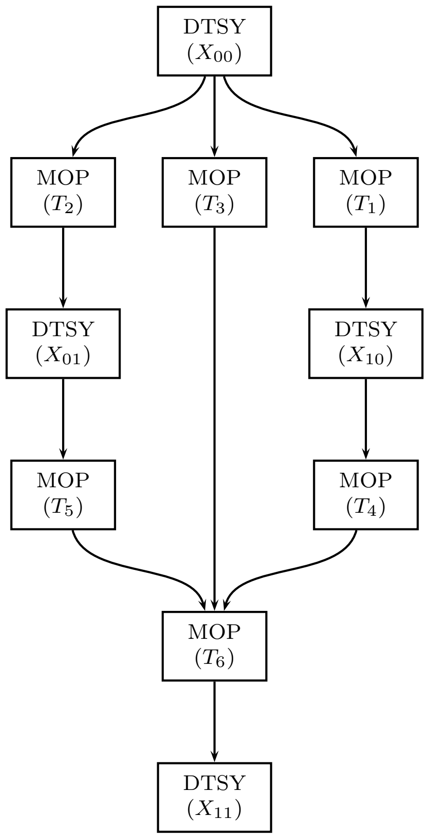


Figure 3.12: Task Graph for DTSY. DTSY boxes are smaller instances of DTSY. MOP boxes are matrix operations

Chapter 4

Evaluation

In this chapter, we will present practical evaluation of Hydra. We first discuss implementation choices we had to make in section 4.1 . Then we present StarPU, the runtime system we chose to generate code within in section 4.2. Section 4.3 shows the effort required to use Hydra, both for the user and the system it runs on. Finally section 4.4 presents our experimental setup and protocol and section 4.5 the experimental results.

4.1 Implementation Choices

To evaluate Hydra, we had a few implementation choices to make within the code generator.

First, we selected StarPU [3] as our back-end. StarPU offers both a user friendly application programming interface (API) to express task parallelism and a runtime system to handle the scheduling of those tasks. We also make use of a few StarPU functionalities in order to improve the quality of the generated code as well as limiting the impact of some weaknesses in our approach in its current state.

An important feature of StarPU for Hydra is that it allows, through the use of tagged tasks, to specify random graphs in a very straightforward fashion and is not limited to fork/join type paradigms.

As discussed in chapter 3 (section 3.3.2) when resorting to canonical signatures for identification, an expansion stage is necessary that introduces multiple new matrices to contain intermediary matrices. To limit the negative impact on memory, we used the concept of lazy registration provided by StarPU. This allowed us not to have to allocate the memory directly, letting the system do it at the latest possible moment at execution. Another feature that would be very useful to deal with such matrices would be garbage collection, relying on the runtime system to also release the memory once the matrix becomes obsolete.

The other major implementation choice we had to make concerned what to use to perform the actual computation in the nodes of the graphs. We chose to use the Intel MKL [15], a state of the art library hand tuned at the assembly level for Intel architectures that provides routines for all the problems we looked at. As a bonus (see section 4.4) it provides sequential and parallel routines for most problems, which also gave us a comparison point.

4.2 StarPU

StarPU is a runtime system that offers support for heterogeneous multicore architectures. It not only offers a unified view of the computational resources (i.e. CPUs and accelerators at the same time), but it also takes care of efficiently mapping and executing tasks onto an heterogeneous machine while transparently handling low-level issues such as data transfers in a portable fashion.

One of the StarPU primary data structures is the codelet. A codelet describes a computational kernel that can possibly be implemented on multiple architectures such as a CPU, a CUDA device or a Cell's SPU.

Another important data structure is the task. Executing a StarPU task consists in applying a codelet on a data set, on one of the architectures on which the codelet is implemented.

A task thus describes the codelet that it uses, but also which data are accessed, and how they are accessed during the computation (read and/or write). StarPU tasks are asynchronous: submitting a task to StarPU is a non-blocking operation. The task structure can also specify a callback function that is called once StarPU has properly executed the task. It also contains optional fields that the application may use to give hints to the scheduler (such as priority levels).

By default, task dependencies are inferred from data dependency (sequential coherence) by StarPU. However in this work, the dependencies are declared explicitly through tagging of all tasks.

4.3 User and Machine Effort

In this section, we evaluate the effort required to generated code for LU decomposition.

User effort: The user must write the problem description (As shown in figure 3.6). The user must also provide a kernel that solves that problem sequentially. In our case, the user is also required to provide the definition of a codelet to connect the kernel to the runtime system. Figure 4.1 describes the user input for a matrix multiplication in Hydra. Figure 4.1a shows the codelet description and figure 4.1b the code of the wrapper necessary to use an MKL library call within StarPU. Chapter A contains the code for all codelets used evaluating Hydra.

Machine effort: The full process for sizes ranging from 1000 to 12,000 by increments of 1000, evaluating for 3 different tiling factors took 21 minutes; and for 5 it took 1 hour. Table 4.1 presents the time spent generating different versions and compiling them for different tiling factors.

```
struct starpu_codelet gemm_cl = {
 .where = STARPU_CPU,
 .cpu_funcs = {__gemm, NULL},
 .nbuffers = 3,
 .modes = {STARPU_R, STARPU_R, STARPU_RW}
};
```

(a) Codelet Declaration

```
void __gemm(void *buffers[], void *cl_arg)
{
 struct params *params = cl_arg;
 int n = params->n;

 double *a = (double *)
 STARPU_MATRIX_GET_PTR(buffers[0]);
double *b = (double *)
 STARPU_MATRIX_GET_PTR(buffers[1]);
double *c = (double *)
 STARPU_MATRIX_GET_PTR(buffers[2]);

 cblas_dgemm(CblasRowMajor,
 CBlasNoTrans, CBlasNoTrans,
  n, n, n,
  1.0,
  a, n, b, n,
  1.0,
  c, n
 );
}
```

(b) Kernel Code

Figure 4.1: User Code

Blocks	5	8	10	16	20
Generation	1 sec.	2 sec.	6 sec.	1 min. 20 sec.	4 mins.
Compilation	1 sec.	5 sec.	10 sec.	2 mins.	7 mins.

Table 4.1: Version Generation and Compilation

4.4 Experimental Setup

All our experiments were conducted on a 32-core (64 threads) platform composed of four 8-core Intel L7555 CPUs (Nehalem) with 64GB of memory.

We are using the Intel C Compiler package version 12.1.3 including the Intel MKL and StarPU version 1.0.

All our experiments were led through repetitive execution and measurement of every version then statistical analysis of the results since some random factors can interfere with the experiments and lead to results not representative of the efficiency of the code tested.

We discard bad outliers, measured times that are too slow and far from the average. On systems where frequency scales to match the load, the first couple of executions of the tested solution also end up serving as warm ups of sorts, signals to the processor to ”wake up” if it had entered a power saving mode and scaled down the frequency.

4.5 Results

Starting from the description of different linear problems on matrices, Hydra automatically generates parallel programs solving these problems. It does so by creating task graphs that use existing sequential kernels in the computation nodes. Those task graphs are scheduled dynamically by the StarPU runtime system.

The Intel MKL [15] (Math Kernel Library) is a state of the art library that includes all the BLAS routines necessary for the problems we studied, hand tuned at the assembly level for Intel's architectures. It includes both sequential and parallel implementations of most of those routines. To evaluate the quality of the versions generated by Hydra, we use MKL's sequential kernels to build our graphs as well as a sequential baseline to measure the speed up achieved through our parallelization. We then compare our best parallel version with the MKL's parallel routine. The problem size is a key factor in performance for dense linear algebra, we thus perform experiments on a range of sizes.

Figure 4.2 presents the results for matrix multiplication. Here the decomposition obtained through Hydra corresponds to a blocked matrix multiplication. It is important to note that it is not an in-place blocked matrix multiplication since Hydra generates copies for each tile. The individual matrix multiplications on the tiles are performed on tiles that occupy continuous space in memory. Locality is improved without the cost of memory strides at the end of lines. We observe that the best parallel code generated by Hydra consistently outperforms the MKL parallel version of the matrix multiplication, for matrix sizes over

Tiling factor	Tasks	Graph Breadth	Copies
2	12	8	12
4	112	64	48
5	225	125	75
8	960	512	192
10	1900	1000	300
16	7936	4096	768

Table 4.2: Matrix Multiplication: Version Characteristics

4000. Table 4.2 shows characteristics of the different versions generated by Hydra. Each version is identified by the tiling factor applied to all the operands. The first column gives the total number of tasks created, the second the breadth of the graph and finally the number of copy tasks.

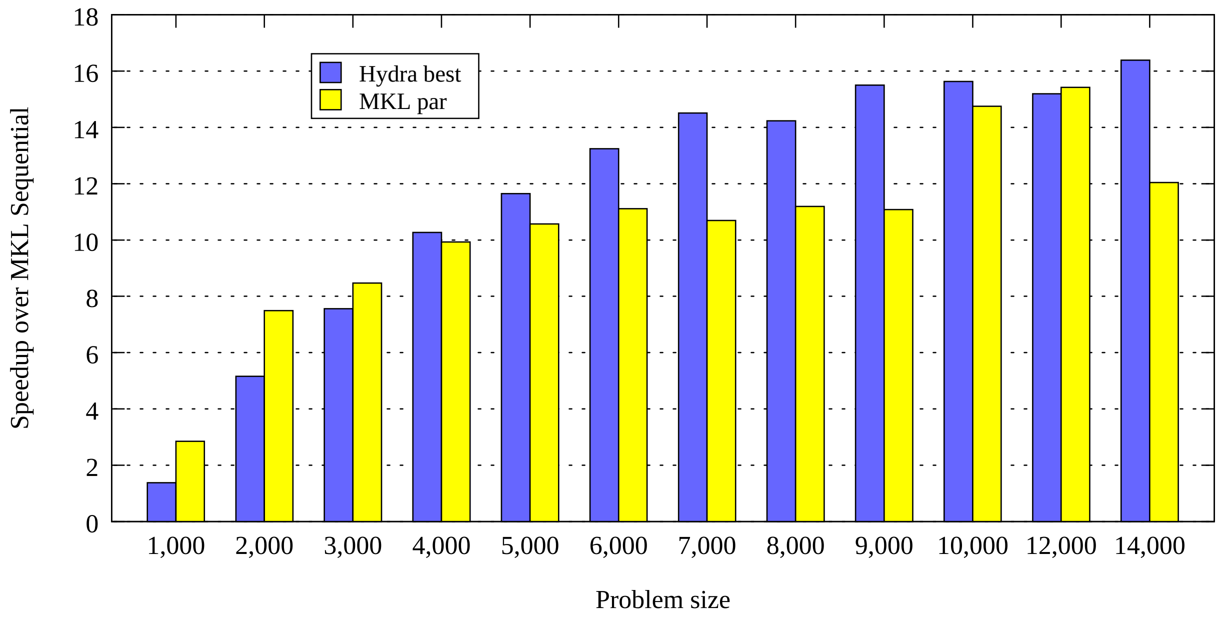


Figure 4.2: Matrix Matrix Multiplication: X = A*B

Figure 4.3 presents performance speed-ups for Hydra and parallel MKL compared to the sequential MKL corresponding routine for the triangular solver. The triangular solver is the matrix form of a triangular system of equations with multiple right hand sides $L{\cdot}X = C$ (with L a lower triangular matrix and X the unknown). Performance of Hydra generated codes

Tiling factor	Tasks	Graph Breadth	Copies
2	8	2	11
4	64	12	42
5	125	20	65
8	512	56	164
10	1000	90	255
16	4096	240	648

Table 4.3: Triangular Solver: Version Characteristics

remains within roughly 10% of the parallel MKL performance. A more detailed analysis in figure 4.4 shows the influence of the number of blocks on performance. We can see that in general, 10 by 10 tiling yields the best results, or a close second. We can also observe that for this particular routine, using a tiling factor that creates tiles of size 1000 by 1000 makes a big difference. This is easily observable for sizes $10,000$ and $16,000$. We can also observe that over decomposition is not always a good strategy.

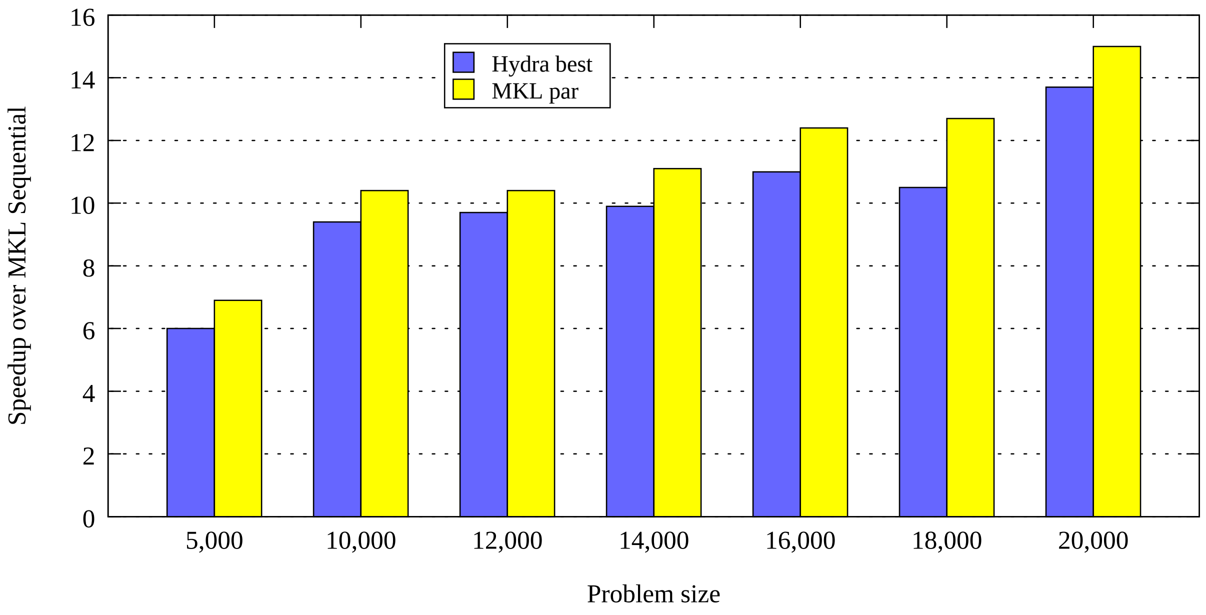


Figure 4.3: Triangular Solver: L*X = C

Table 4.3 shows that for a tiling factor of 10, there are 1000 tasks created, and the maximum number of tasks executable at the same time during the course of the execution is

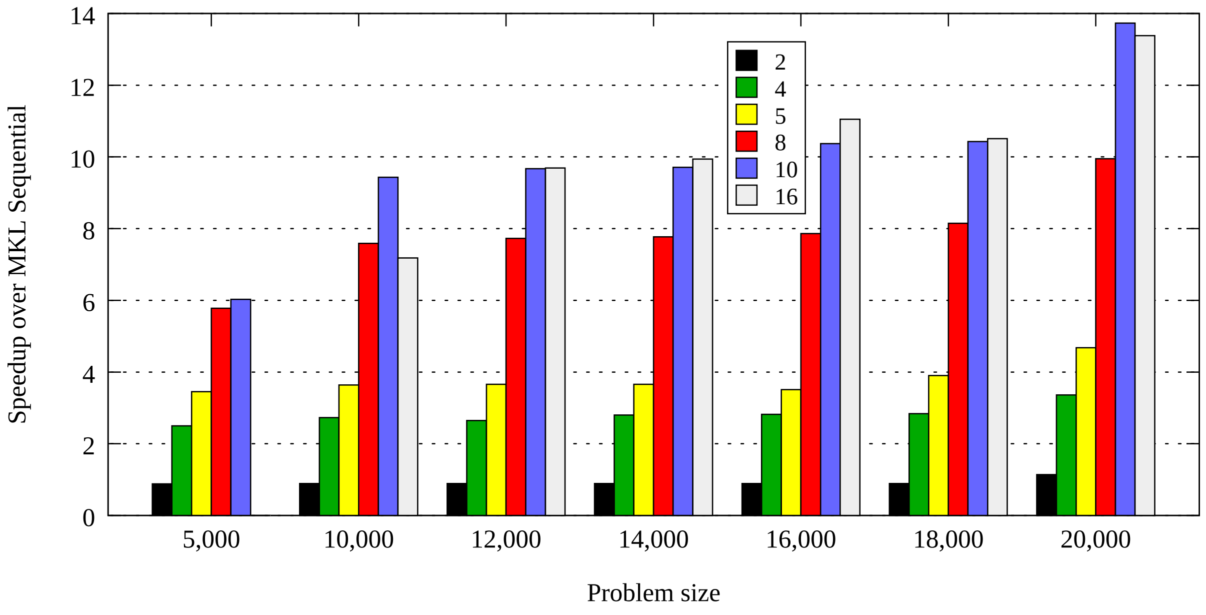


Figure 4.4: Exploring Tiling Factors for the Triangular Solver: L*X = C.

90 (this is the breadth of the graph) and 255 copies of tiles are performed. The high number of copies compared to the number of computational tasks may account for some performance loss. Another factor to take into account are the hidden copies. We call a hidden copy, a copy that occurs within a kernel and not as a task in the generated code. Figure A.4 in Appendix A shows the code of the kernel used for the triangular solver within a Hydra generated code. Due to the difference of semantics followed in this version of Hydra and that of the MKL, namely the MKL overwrites on of the inputs with the result, there is the need for a data copy within the kernel.

For the Continuous Triangular Sylvester Equation (CTSY), figure 4.5 compares the speed ups achieved by Hydra's best generated version to that of the parallel MKL, both over the sequential routine of the MKL. The speed-up over 40 achieved on a 32 core machine can be explained by the fact that Hydra decomposed the original problem into sub problems that have higher sequential efficiency than the MKL's implementation, namely, general matrix multiplications and daxpy operations. The speed up thus comes from both parallelism and from the use of more efficient kernels. This aspect of Hydra is discussed further in chapter 5.

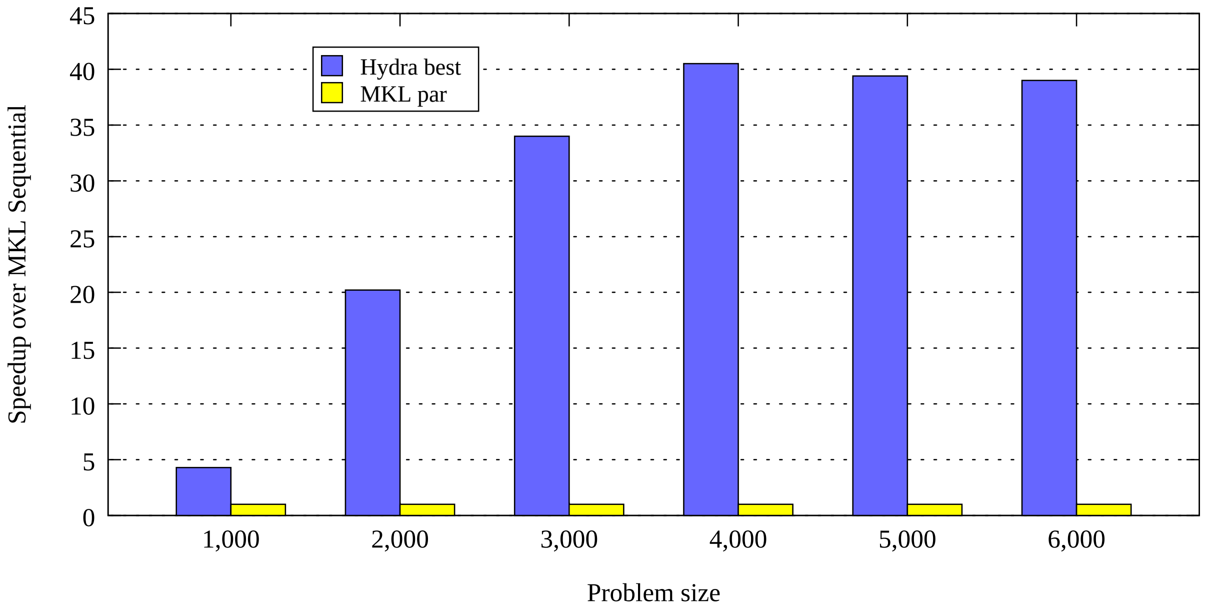


Figure 4.5: Triangular Sylvester: $AXB - X = C$

Tiling factor	Tasks	Graph Breadth	Copies
2	12	2	14
4	112	12	52
5	225	18	80
8	960	36	200
10	1900	48	310
16	7936	97	784

Table 4.4: Continuous Triangular Sylvester (CTSY): Version Characteristics

The task graph obtained for a 2 by 2 tiling of CTSY is shown in figure 4.6. Rectangular tasks are copy tasks, darker rounded tasks are smaller instances of CTSY and the others are various BLAS-3 operations (matrix additions, subtractions and multiplications). The algorithms automatically generated by Hydra to solve CTSY correspond to the method described by Jonsson *et al.* [18].

Table 4.4 presents the characteristics of the versions generated by Hydra for CTSY.

Moreover, we observe that there is no parallel implementation of CTSY in the MKL. This is a practical example of the benefit of Hydra, from the existing sequential kernel, we were able to automatically generate, with little to no user effort, a parallel version of this problem

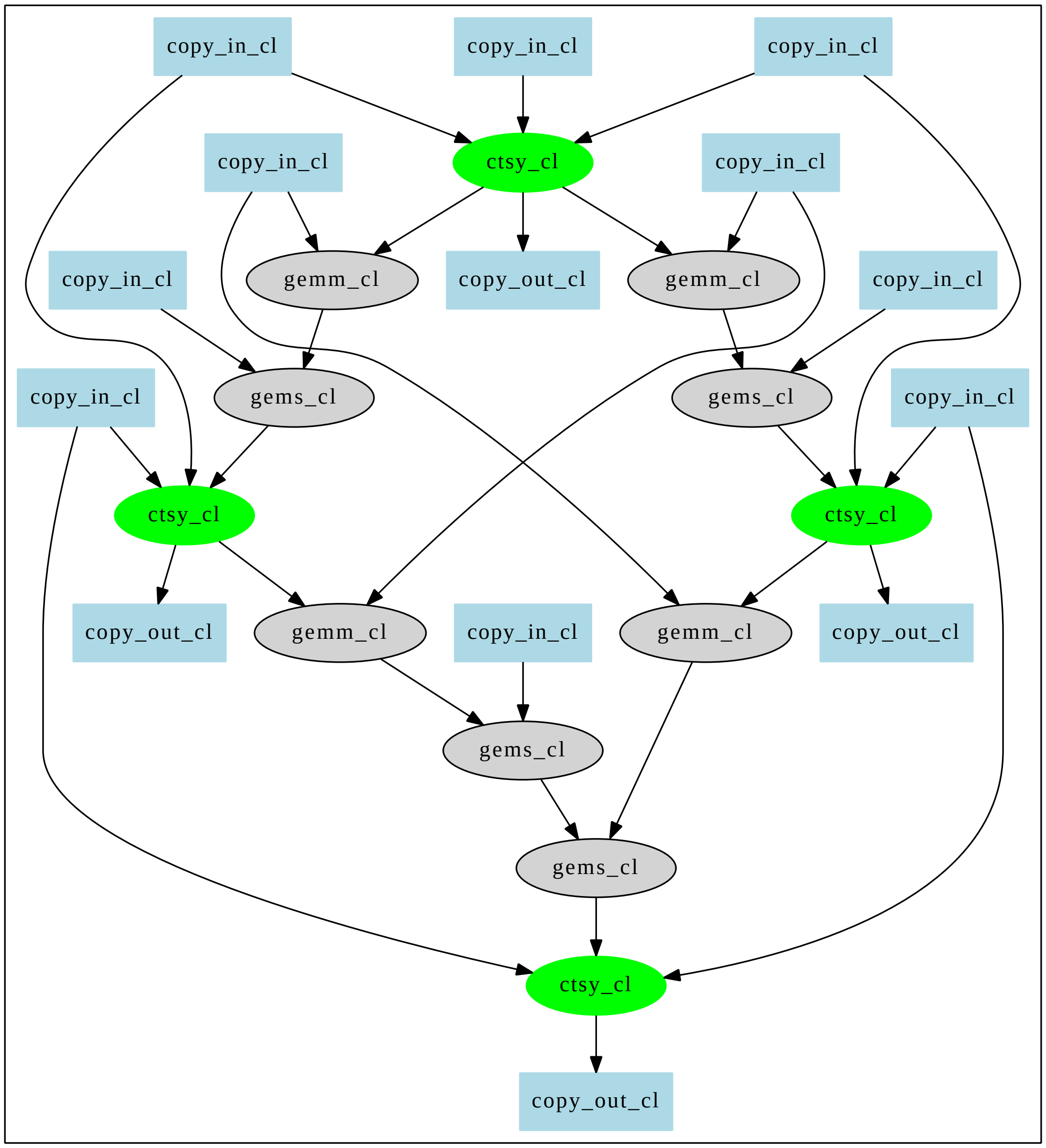


Figure 4.6: CTSY Task Graph for 2 by 2 Tiling

Tiling factor	Tasks	Max Parallelism	Copies
2	6	2	10
4	44	9	36
5	85	16	55
8	344	49	136
10	670	81	210
16	2736	225	528

Table 4.5: LU Decomposition: Version Characteristics

that achieved good performance (A speed up of forty two on a thirty two core machine).

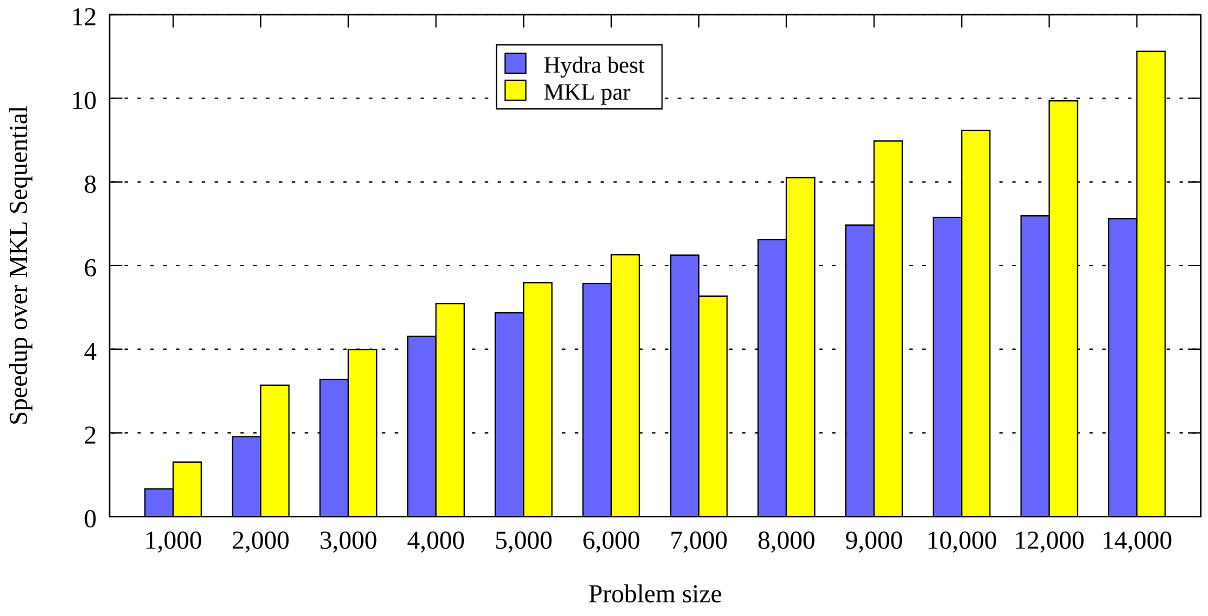


Figure 4.7: LU Factorization: L*U = A

Finally, Figure 4.7 shows performance for LU factorization. While the parallel MKL LU outperforms the code generated by Hydra, we notice that at 7000 the trend is reversed, just before Hydra's performance flatlines. This can be explained by the extra copies in the LU kernel. (Figure A.6)

Chapter 5

Extensions to Hydra

Parallel frameworks are not the only possibilities for Hydra backends. The generated graph also has interesting properties for sequential output. [12] Flattening of the task graph leads to sequential algorithms. Such algorithms can be used to improve locality (see section 5.1) or reduce program complexity (section 5.2). The methodology used in Hydra and described in this thesis can also be applied to build specialized systems (section 5.3) for problems that cannot be directly solved by Hydra such as the QR decomposition (section 5.4).

5.1 Improved Locality

An effect that can be expected from decomposing the problem the way Hydra does then flattening the graph is improved data locality. Moreover, the reshaping of the data ensures that every tile is made of consecutive memory, theoretically improving the called kernel's performance. On selected examples, slowdown was observed on random flattening of the generated graph. Which leads us to realize that just like with parallel programs, scheduling of the computational nodes is neither trivial nor unimportant in sequential execution.

5.1.1 Projected Bounds on Sequential Performance

To evaluate the potential for speedup with sequential execution of hydra generated graphs, we independently benchmarked the different kernels used to rebuild the problems studied and tallied the kernels used in the different graphs to build a projected execution time that disregards any overhead. This gives us an upper bound to the achievable speedup.

As for parallel execution, we use the Intel MKL as our core library. We are thus applying and extra outer level of tiling to MKL kernels.

Tables 5.1, 5.2 and 5.3 present projected speedups (in %) over sequential MKL for Hydra decomposition algorithms.

Tiling	Size	GEMM	GEMA	COPY	Reference Time	Projected Time	Projected Speedup
2	1000	8	4	12	5.40E+08	6.16E+08	87.58
	2000				4.21E+09	4.43E+09	94.97
	4000				3.33E+10	3.40E+10	97.88
	8000				2.66E+11	2.68E+11	99.23
	10000				5.19E+11	5.21E+11	99.51
4	1200	64	48	48	9.24E+08	1.38E+09	67.15
	2000				4.21E+09	4.97E+09	84.55
	4000				3.33E+10	3.54E+10	94.11
	8000				2.66E+11	2.72E+11	97.75

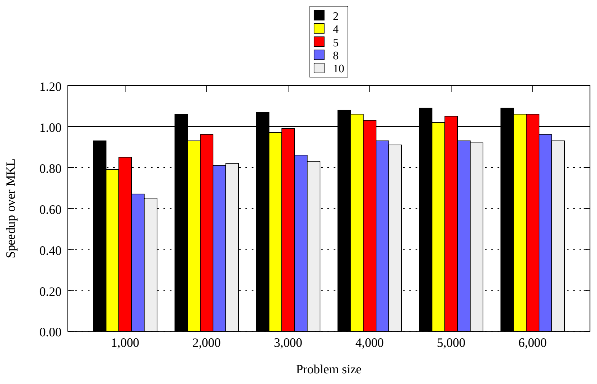
Table 5.1: GEMM: Projected Speedups for Hydra Decomposition of Matrix Product

Tiling	Size	GEMM	GEMA/S	TriSlv	COPY	Ref. Time	Proj. Time	Proj. S/U
2	1000	2	2	4	11	5.40E+08	3.64E+08	148.11
	2000					4.21E+09	2.46E+09	170.71
	3000					1.41E+10	7.86E+09	179.43
	4000					3.33E+10	1.81E+10	183.86
4	1200	24	24	16	42	9.24E+08	7.79E+08	118.64
	2000					4.21E+09	2.71E+09	155.15
	3200					1.71E+10	9.96E+09	171.64
	4000					3.33E+10	1.87E+10	178.25

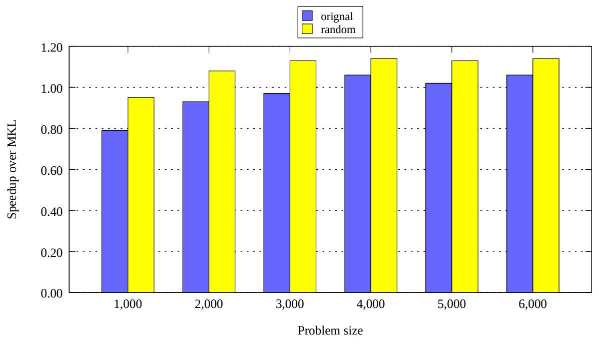
Table 5.2: TriSlv: Projected Speedups for Hydra Decomposition of Triangular Solver

Tiling	Size	GEMM	GEMA GEMS	TrSlv	LU	COPY	Ref. Time	Proj. Time	Proj. S/U
2	1000	1	1	2	2	10	5.40E+08	2.54E+08	212.47
	2000						4.21E+09	1.67E+09	251.27
	3000						1.41E+10	5.31E+09	265.74
	4000						3.33E+10	1.22E+10	273.35
4	1200	14	14	12	4	36	9.24E+08	5.30E+08	174.28
	2000						4.21E+09	1.83E+09	229.60
	3200						1.71E+10	6.70E+09	255.10
	4000						3.33E+10	1.25E+10	265.37

Table 5.3: LU: Projected Speedups for Hydra Decomposition of LU Factorization



(a) LU: Sequential Tiling, Direct Flattening



(b) LU: Comparison of Direct Flattening with Random Scheduling for 4 by 4 Tiling

Figure 5.1: Experimental Measurements of Sequential LU

Figure 5.1a presents the effective speed ups observed for LU with direct flattening of the task graph. We can see that there is little to no improvement over the MKL alone. However, figure 5.1a shows the impact of scheduling even for sequential execution. It presents two different schedulings side by side, for a 4 by 4 tiling, the original scheduling and a randomly generated one. Here the original scheduling is the one created by our deterministic flattening algorithm.

5.2 Reduced Program Complexity

One of the reasons for developing BLAS in the beginning was to optimize algorithms and programs by decomposing or rewriting them incorporation BLAS operations. BLAS operations could then be independently optimized and packed within libraries and impact a wide array of problems simultaneously. It is trivial to observe that with the rules described for Hydra, the generated algorithms are expressed with multiple matrix multiplications and additions, which can be translated into gemm and daxpy kernel calls, two kernels that are highly optimized in the MKL.

CTSY and DTSY are not main problems of interest and thus not as much effort is put in their optimization [14]. By decomposing such a problem with Hydra, many BLAS operations are exposed in the new algorithm, leading to increased performance. Figure 5.2

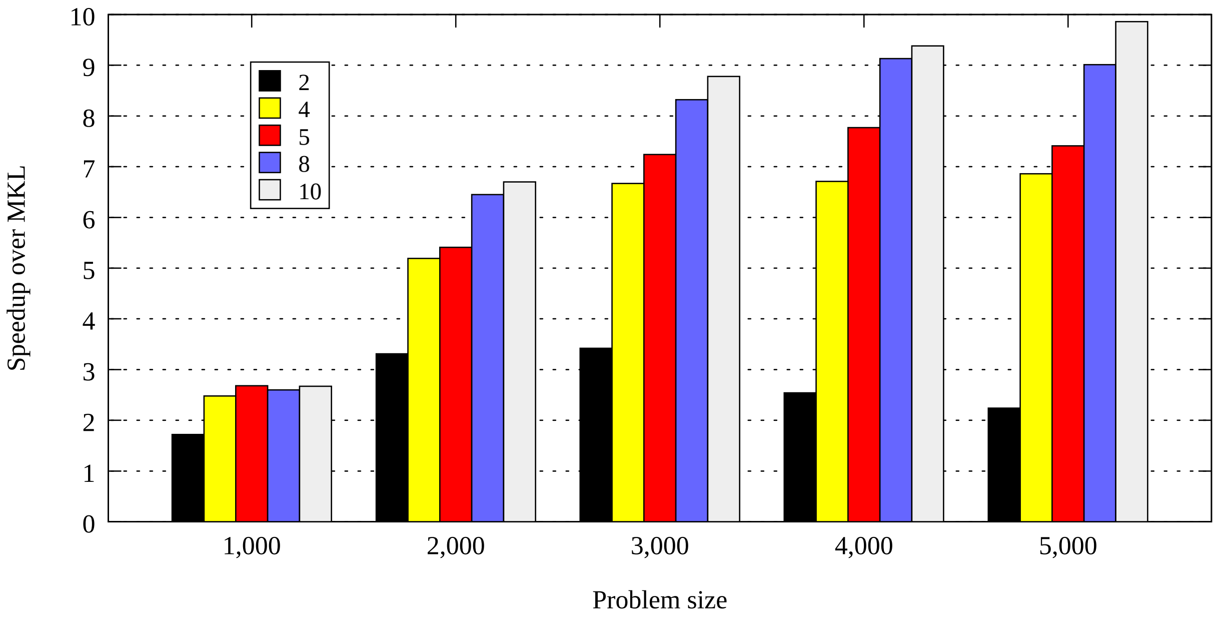


Figure 5.2: CTSY: Random Schedule Sequential Speedup

5.3 Building Specialized Systems

Conceptually, the approach behind Hydra is not linked to traditional tiling and Basic matrix operations like presented in chapter 3 and evaluated in chapter 4. The process can be decomposed in two different steps. First, the application of satisfiability rules to identify valid solutions and define the search space. Second the generation of solutions using a set of rewriting rules. From an engineering point of view, the system can be built in a modular way that would allow the user to specify their own set of core rules for application of our divide and conquer and forward substitution approach.

Different sets of rules would lead to different application scope. Hydra could thus be extended to different domains of linear algebra through the definition of new rules. Another possibility is to use Hydra as a framework to build specialized autotuning systems. When considering a problem with known decomposition techniques, a specialized system can be built, with problem specific rules. Section 5.4 describes this process in the case of the QR decomposition.

5.4 Generating QR Solvers

The QR decomposition is an example of an application of our divide and conquer approach using different rules. In this case, knowledge of Householder reflection matrices and their properties are required by the system to solve the problem at hand.

Problem description: the QR decomposition (5.1) is the decomposition of a matrix (A) into the product of an orthogonal matrix (Q) and an upper triangular matrix (R).

$$A = Q \cdot R \tag{5.1}$$

Householder Reflection Matrices

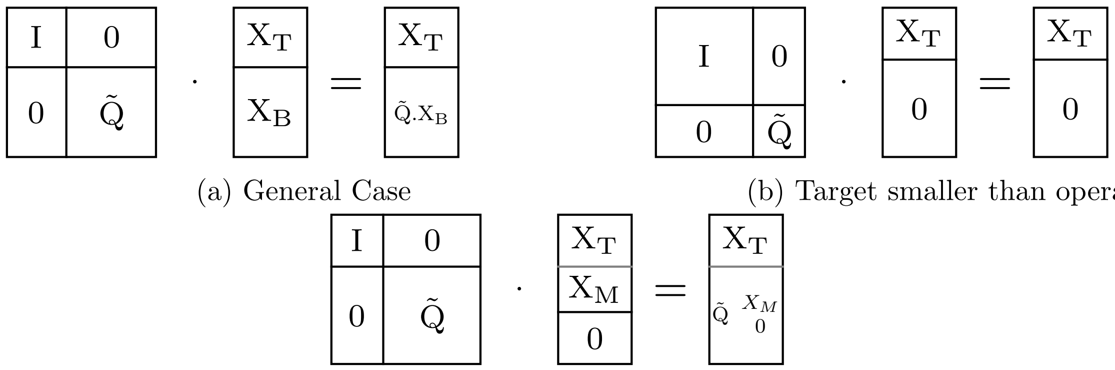
A Householder reflection matrix can be built to zero all elements of a vector except for one. Equation (5.2) illustrates the use of such a matrix. Another property of Householder matrices is that they are orthogonal.

$$Q \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \alpha_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{5.2}$$

There is a function that applied to any vector, will produce a matrix Q that behaves as depicted in (5.2). Equation (5.3) illustrates how to build a matrix to put zeros in the lower part of a vector.

$$\begin{gathered} \begin{bmatrix} I & 0 \\ 0 & \tilde{Q} \end{bmatrix} \cdot \begin{bmatrix} v_T \\ v_B \end{bmatrix} = \begin{bmatrix} v_T \\ \tilde{Q} \cdot v_B \end{bmatrix} \\ with\ \tilde{Q} \cdot v_B = \begin{bmatrix} v_B^{'} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \end{gathered} \tag{5.3}$$

Where $\tilde{Q}$ is built for v_B as in (5.2). Figure 5.3 shows the effects of multiplying a Householder matrix with a vector (or matrix columns) that it is not designed to affect. As figure 5.3b shows, multiplying an already zeroed column by a Householder matrix that targets a lower portion of the matrix has no effect. Thus Householder matrices can be used to triangularize matrices as follows:



(a) General Case (b) Target smaller than operand

(c) Target larger than operand

Figure 5.3: Multiplication by Householder matrices

$$\begin{cases} Q_1 \cdot \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} \alpha_{11} & a'_{12} & a'_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & a'_{32} & a'_{33} \end{bmatrix} \\ Q_2 \cdot \begin{bmatrix} \alpha_{11} & a'_{12} & a'_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & a'_{32} & a'_{33} \end{bmatrix} = \begin{bmatrix} \alpha_{11} & a'_{12} & a'_{13} \\ 0 & \alpha_{22} & a"_{23} \\ 0 & 0 & a"_{33} \end{bmatrix} \end{cases}$$

And the following equation holds :

$$Q_2 \cdot Q_1 \cdot A = R$$

A known property of orthogonal matrices is that their inverse and their transpose are the same, and are themselves orthogonal.

$$(Q_2 \cdot Q_1)^T \cdot R = A$$

Providing the solution to the QR decomposition of A.

Derivation of QR Decomposition

In this section, we illustrate a derivation of QR within our framework.

$$Q \cdot A = R \tag{5.4}$$

First, using Householder's properties, Q is divided into three instances that each target a different slice of A.

$$Q_R \cdot Q_M \cdot Q_L \cdot A = R \tag{5.5}$$

We define zero tiling as the decomposition of a matrix as the decomposition of a matrix as a sum of matrices containing a set of columns of the original matrix and zero elements everywhere else.

Zero tiling is applied on A and R matching the targets of the Householder matrices.

$$Q_R \cdot Q_M \cdot Q_L \cdot (A_L + A_M + A_R) = R_L + R_M + R_R \tag{5.6}$$

The problem is now divided into 3 equations.

$$\begin{cases} Q_R \cdot Q_M \cdot Q_L \cdot A_L = R_L \\ Q_R \cdot Q_M \cdot Q_L \cdot A_M = R_M \\ Q_R \cdot Q_M \cdot Q_L \cdot A_R = R_R \end{cases} \tag{5.7}$$

Using the rules described in figure 5.3 products with no side effects are eliminated, producing the following simplified system:

$$\begin{cases} Q_L \cdot A_L = R_L \\ Q_M \cdot Q_L \cdot A_M = R_M \\ Q_R \cdot Q_M \cdot Q_L \cdot A_R = R_R \end{cases} \tag{5.8}$$

Work on the equations' signatures as illustrated in section 3.3.2 leads to the following system.

$$\begin{cases} Q_L \cdot A_L = R_L & (5.9a) \\ Q_L \cdot A_M = A_M^{'} & (5.9b) \\ Q_M \cdot A_M^{'} = R_M & (5.9c) \\ Q_L \cdot A_R = A_R^{'} & (5.9d) \\ Q_M \cdot A_R^{'} = A_R^{"} & (5.9e) \\ Q_R \cdot A_R^{"} = R_R & (5.9f) \end{cases}$$

Figure 5.4 shows the task graph generated from this system. In this graph, boxes labelled as QR are smaller instances of QR. In this case smaller doesn't mean smaller matrices, but rather smaller scope. The QR subproblems become localized problems targeting small collections of columns rather than full matrices. MOP boxes represent basic matrix operations such as multiplications, additions or subtractions.

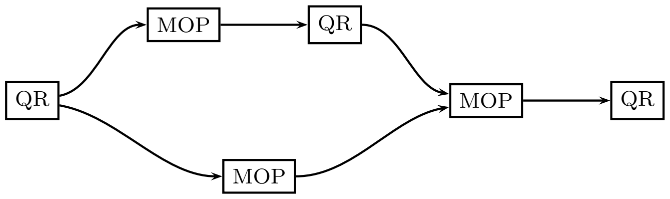


Figure 5.4: Task Graph for QR

Similarly to LU in chapter 4 the parallelism doesn't appear between instances of QR itself but rather in matrix operations that allow the problem to be decomposed. Higher degrees of tiling would expose additional parallelism. A similar system can be built using Givens rotation matrices, another known method to solve the QR problem. It might also be possible to combine both and find new algorithms.

Manually following this sequence in MATLAB [20], we were able to verify that the sequence of equations is a valid solution to QR. However, time constraints prevented us from building this system and empirically verifying. Those results are however encouraging that Hydra could be extended in the future to encompass more problems.

Chapter 6

Discussion and Conclusions

In this chapter, we will talk about the limitations to Hydra in its current state in section 6.1. We will then discuss our results and possible future directions for this research in section 6.2 before concluding this thesis in section 6.3.

6.1 Limitations

Within the general framework described for Hydra, there are a few limitations to the scope of problems that can be automatically derived. The first comes from the way we derive new equations through symbolic execution of the tiled operands and then build dependencies. For that approach to work, it is necessary that all operands, whether matrices or vectors, be only manipulated as a whole. This particular limitation precludes stencil problems from being derived in Hydra such as it is currently specified. However, many stencil problems can be tiled and solve hierarchically [29]. The methodology used in Hydra to expose parallelism in problems can thus be conceivably be applied to stencil problems. Extending the expressibility of the input language and finding how to deal with the necessary overlapping tiles could present an interesting new avenue of research for Hydra.

The other limitation comes from the tiling itself. The main assumption with this work is that we can tile the problem and after a step of symbolic execution, identify sub problems that if solved in the proper sequence, will contribute to solving the bigger problem. This can only be achieved if the properties of the matrices can “survive” tiling. In the case of dense matrices with no particular property, tiling generates similar dense matrices. On symmetric matrices, tiling will produce symmetric tiles on the diagonal and dense tiles off diagonal; moreover, the tiles in the upper section of the tile are the transpositions of the tiles in the lower half. This fact can be used to simplify generated equations since there is no need to solve for both a tile and its transpose. Triangular matrices, once tiled, present triangular tiles on the diagonal, dense tiles on one side and zero blocks on the other. While diagonal matrices are tiled with diagonal tiles on the diagonal and zero tiles everywhere else. It is essential to the identification process that some of the tiles retain the properties of the original matrices, to allow identification of the original problem in the derived candidates.

Equations that require matrices with properties that do not verify this requirement such as orthogonal or jacobian matrices, are not within the application scope of Hydra. Although we discussed in sections 5.3 and 5.4 how in some cases the general methodology can still be applied.

6.2 Discussion

It is important to note that the results presented to evaluate the approach were obtained on a prototype and intended as a proof of concept. There are many improvements that can be made to the system in its current form that are purely engineering matters.

Hydra is currently following directly the semantics guided by the equations. For example, when looking at the LU decomposition, Hydra will generate code that produces a matrix L and a matrix U. Most BLAS libraries though, including the Intel MKL, overwrite the input matrix A with the result. Since we are using such a library, our kernels must copy the data after calling the library routine (See figure A.6 in appendix A). This results in suboptimal performance. Hydra could be modified to allow overwriting of operands. This would improve the quality of the generated code by eliminating those copies and by lowering their memory footprint.

Memory usage, in particular due to the intermediate matrices generated in the expansion stage, is another point where Hydra could be improved. Given how they are generated, intermediate matrices are only "live" along a single arc of the graph. Special tasks could thus be generated at the consumer nodes to free those variables. Another solution would be to rely on the system having good garbage collection, possibly with the ability to release data in the application.

Extension of Hydra to heterogeneous system, given the runtime system currently targeted, would be a matter of getting a GPU expert to write the appropriate kernels and codelets since StarPU automatically supports heterogeneous systems and handles data movements transparently. New tasks to junction between execution paths of different granularity in the graphs would have to be devised.

Another possible extension for Hydra would allow to handle cases where there is no available sequential implementation. The approach is inherently recursive, the generated graphs could be used to generate recursion patterns to solve the problems within the computation nodes. This would require a new module to create equation solvers in the scalar cases, a problem that is rather straight forward, or require the user to write code solving the scalar case of the presented problem.

Another domain where Hydra's methodology could be applied is for sparse computation. Existing algorithms, such as SPIKE [21], have hierarchical in nature and parameterizable. Sparse algebra is a domain in which further research could be conducted to extend Hydra's scope of applicability.

6.3 Conclusions

Hydra is a parallel code generator for a class of linear algebra problems. It starts from the high-level expression of the equation to solve and generates multiple versions of parallel task graphs solving the problem, for multi-core architectures. The essential idea of Hydra is to use a divide-and-conquer approach to find an algorithmic solution to the initial description of the problem. While the recursive decomposition could lead to scalar problems, we choose to rely on existing highly optimized sequential libraries for the resolution of small enough problems. Moreover, we resort to dynamic scheduling techniques in order to avoid load balancing issues.

We have shown that this approach is able to generate parallel codes with little to no development effort: the user only needs to specify the equation to solve and provide sequential kernels. In some cases some level of interactive development may be required when the discovered algorithms make use of sub problems others than the original and the ones included with Hydra. Moreover, following an auto-tuning approach, the multiple versions generated by Hydra are combined into a code with performance comparable to those of Intel parallel MKL functions, even outperforming it for matrix multiplication on larger sizes. Furthermore Hydra provides parallel implementations for problems with no parallel implementation in MKL such as with Sylvester triangular system resolution the parallel functions of Intel MKL library.

With the currently implemented set of derivation rules, Hydra can handle equations that show recursivity when our divide and conquer approach is applied. But the general methodology can be applied to a wider array of problems, provided appropriate derivation and identification rules are defined. Or be used for faster development of problem specific auto tuners. As we illustrated with the problem of QR factorization. Many problems in different domains can be and have been solved hierarchically. For such problems, Hydra provides a framework and methodology to autotune for parallelism if the user provides the rules necessary for decomposition and derivation.

Future prospects for Hydra can include the generalization of the system to include the generation of parallel codes for heterogeneous architectures. Indeed, one advantage of using a dynamic scheduler such as StarPU [3] is its capacity to handle systems with both CPUs and GPUs. The decision of whether to run the kernel on a CPU or an accelerator is made by the runtime system. The runtime also handles all necessary data transfers. It only requires to provide CPU and GPU versions for all kernels (for instance MKL [15] and PLASMA [13] libraries). Moreover, Hydra offers the opportunity to generate parallel task graphs with non-uniform granularity, through different tile sizes. Such graphs would then have coarser grain execution paths biased towards GPU execution and finer grain paths, better suited for multicore execution.

Appendix A

Hydra v1 : Native Codelets

Following are the implementations of the codelets used in v1.0 of Hydra.

```
void __gemm(void *buffers[], void *cl_arg) {

    struct params *params = cl_arg;
    int n = params->n;

    double *a = (double *)STARPU_MATRIX_GET_PTR(buffers[0]);
    double *b = (double *)STARPU_MATRIX_GET_PTR(buffers[1]);
    double *c = (double *)STARPU_MATRIX_GET_PTR(buffers[2]);
    cblas_dgemm(CblasRowMajor,  // Row Major Storage
            CblasNoTrans,       // A is nottransposed
            CblasNoTrans,       // B is not transposed
            n,                  // Number of rows of A
            n,                  // Number of columns of B
            n,                  // Number of columns of A
            1.0,                // alpha (alpha*A*B + beta*C = C)
            a,                  // A
            n,                  // leading dimension of A
            b,                  // B
            n,                  // leading dimension of B
            0.0,                // beta (alpha*A*B + beta*C = C)
            c,                  // C
            n                   // leading dimension of C
            );
}
```

Figure A.1: StarPU Codelet: Matrix Multiplication (MKL Wrapper)

Figure A.1 shows the wrapper for the matrix multiplication. We used the MKL kernel for dgemm (Double precision General Matrix Multiplication). In this case, `alpha` is set to 1.0 and to avoid having to initialize intermediate variables, we make use of the factor `beta`, setting it to 0 to ignore the values in `C`. The StarPU specific code required is limited to casting the buffers and the size parameter.

```
void __gems(void *buffers[], void *cl_arg) {
    struct params *params = cl_arg;
    int n = params->n;

    double *a = (double *)STARPU_MATRIX_GET_PTR(buffers[0]);
    double *b = (double *)STARPU_MATRIX_GET_PTR(buffers[1]);
    double *r = (double *)STARPU_MATRIX_GET_PTR(buffers[2]);

    int i,j;

    cblas_dcopy(n*n,a,1,r,1); // r = a
    cblas_daxpy(n*n, -1.0, b, 1, r, 1);
}
```

Figure A.2: StarPU Codelet: Matrix Subtraction (MKL Wrapper)

Figure A.2 shows the matrix subtraction. Since we ensure that operands are always copied in contiguous areas of memory, we can consider the matrix subtraction as a vector subtraction on the linearized matrices. We can thus make use of the `daxpy` kernel. However, this kernel overwrites one of the inputs, so we need to first perform a copy of the first operand into the result memory space. We can then perform the `daxpy` operation with the factor set to -1. The matrix addition (figure A.3) is identical but for the factor that is then set to 1.

Figure A.4 shows the code for the Lower Triangular solver. Just like with the addition and subtraction, a copy is necessary. The routine used here comes from LAPACK.

Figure A.5 shows the upper triangular system, it requires an additional copy to prepare the data and uses a CBLAS routine with the appropriate transposition flags set.

Figure A.6 shows the LU decomposition. It stands out by the need of “manual” copies for the resulting triangular matrices since the routine produces a single matrix to optimize for space.

```c
void __gema(void *buffers[], void *cl_arg) {
    struct params *params = cl_arg;
    int n = params->n;

    double *a = (double *)STARPU_MATRIX_GET_PTR(buffers[0]);
    double *b = (double *)STARPU_MATRIX_GET_PTR(buffers[1]);
    double *r = (double *)STARPU_MATRIX_GET_PTR(buffers[2]);

    int i,j;

    cblas_dcopy(n*n,a,1,r,1); // r = a
    cblas_daxpy(n*n, 1.0, b, 1, r, 1);
}
```

Figure A.3: StarPU Codelet: Matrix Addition (MKL Wrapper)

Figure A.7 contains the code for Continuous Triangular Sylvester equation. The routine also requires a copy since the routine overwrites one of the inputs.

```
void __trilow(void *buffers[], void *cl_arg) {
    struct params *params = cl_arg;
    int n = params->n;
    int info;

    double *l = (double *)STARPU_MATRIX_GET_PTR(buffers[0]);
    double *b = (double *)STARPU_MATRIX_GET_PTR(buffers[1]);
    double *x = (double *)STARPU_MATRIX_GET_PTR(buffers[2]);

    int i,j;

    cblas_dcopy(n*n, b, 1, x, 1); // x = b

    info = LAPACKE_dtrtrs(
            LAPACK_ROW_MAJOR,
            'L',        // Lower Triangular System
            'N',    // Not transposed
            'N',    // Diagonal elements not assumed to be 1
            n,      // Number of rows in L
            n,      // Number of right hand sides
            l,      // L
            n,
            x,
            n
            );
}
```

Figure A.4: StarPU Codelet: Lower Triangular System (MKL Wrapper)

```
void __trihirev(void *buffers[], void *cl_arg) {
    struct params *params = cl_arg;
    int n = params->n;
    int info;

    double *b = (double *)STARPU_MATRIX_GET_PTR(buffers[0]);
    double *u = (double *)STARPU_MATRIX_GET_PTR(buffers[1]);
    double *x = (double *)STARPU_MATRIX_GET_PTR(buffers[2]);

    cblas_dcopy(n*n, b, 1, x, 1); // x = b

    cblas_dtrsm(CblasRowMajor,
            CblasRight,
            CblasUpper,
            CblasNoTrans,
            CblasNonUnit,
            n,
            n,
            1.0,
            u,
            n,
            x,
            n
            );

}
```

Figure A.5: StarPU Codelet: Upper Triangular System (MKL Wrapper)

```c
void __lu(void *buffers[], void *cl_arg) {

    struct params *params = cl_arg;
    int n = params->n;

    double *a = (double *)STARPU_MATRIX_GET_PTR(buffers[0]);
    double *l = (double *)STARPU_MATRIX_GET_PTR(buffers[1]);
    double *u = (double *)STARPU_MATRIX_GET_PTR(buffers[2]);

    int *perm = (int *)malloc(n * sizeof(int));

    double (*nl)[n] = (double (*)[n])l;
    double (*nu)[n] = (double (*)[n])u;
    double (*na)[n] = (double (*)[n])a;

    int result = LAPACKE_dgetrf(LAPACK_ROW_MAJOR,
            n,
            n,
            a,
            n,
            perm
            );

    int i,j;
    for(i = 0 ; i < n ; i++) {
        for(j = 0 ; j < n ; j++) {
            if(i<=j) {
                nl[i][j] = 0.0;
                nu[i][j] = na[i][j];
            }
            if(i>j){
                nu[i][j] = 0.0;
                nl[i][j] = na[i][j];
            }
            if(i == j) {
                nl[i][j] = 1.0;
            }
        }
    }

    free(perm);

}
```

Figure A.6: StarPU Codelet: LU Factorization (MKL Wrapper)

```
void __ctsy(void *buffers[], void *cl_arg) {

    double alpha = 1.0;

    struct params *params = cl_arg;
    int n = params->n;

    double *l = (double *)STARPU_MATRIX_GET_PTR(buffers[0]);
    double *u = (double *)STARPU_MATRIX_GET_PTR(buffers[1]);
    double *c = (double *)STARPU_MATRIX_GET_PTR(buffers[2]);
    double *x = (double *)STARPU_MATRIX_GET_PTR(buffers[3]);

    cblas_dcopy(n*n, c, 1, x, 1); // x = c

    int result = LAPACKE_dtrsyl(LAPACK_ROW_MAJOR, // Storage order
            'T',    // trana : T for A lower triangular
            'N',    // tranb : N for B upper triangular
            1,      // A*X - X*B or A*X + X*B
            n,      // order of A (number of rows in X)
            n,      // order of B (number of columns in X)
            l,      // A
            n,      // leading dimension of A
            u,      // B
            n,      // leading dimension of B
            x,      // C
            n,      // leading dimension of C
            &alpha  // scale factor
            );

}
```

Figure A.7: StarPU Codelet: CTSY, Triangular Sylvester Equation (MKL Wrapper)

Appendix B

Sample Graphs

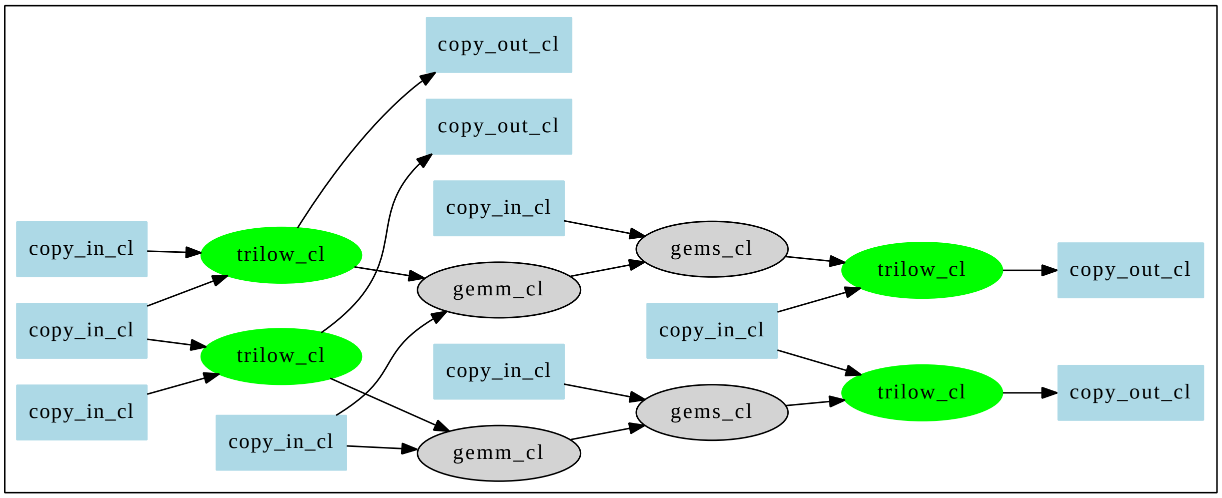


Figure B.1: Triangular Solver: 2 by 2 Tiling

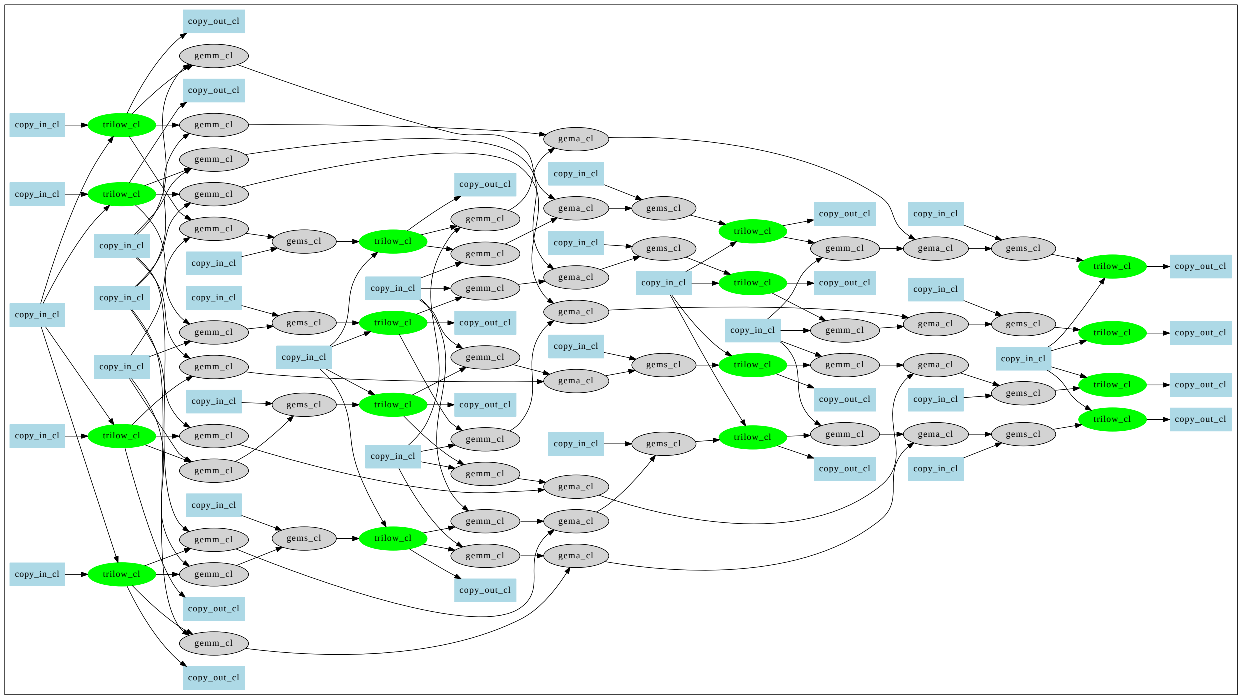


Figure B.2: Triangular Solver: 4 by 4 Tiling

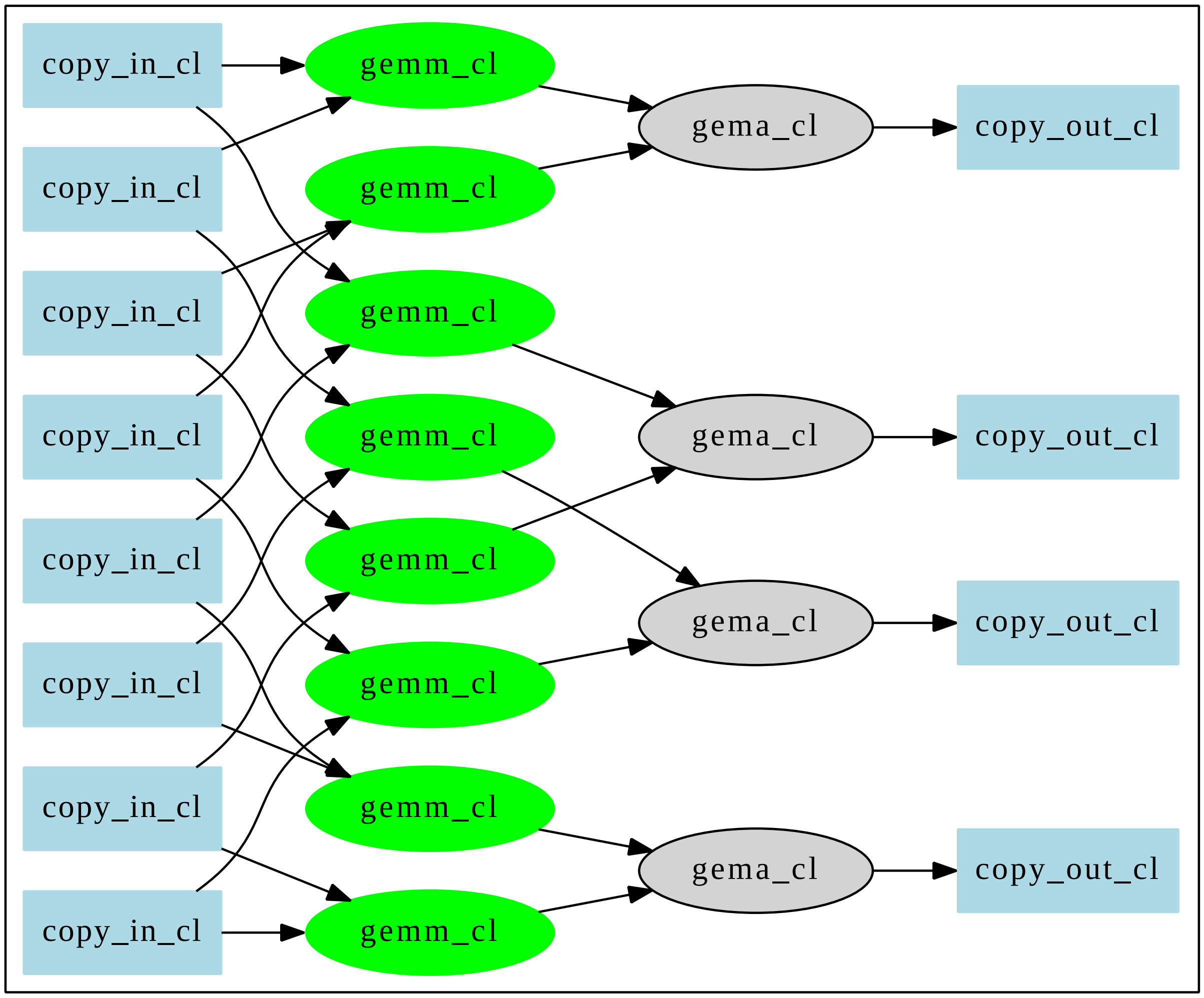


Figure B.3: Matrix Multiplication: 2 by 2 Tiling

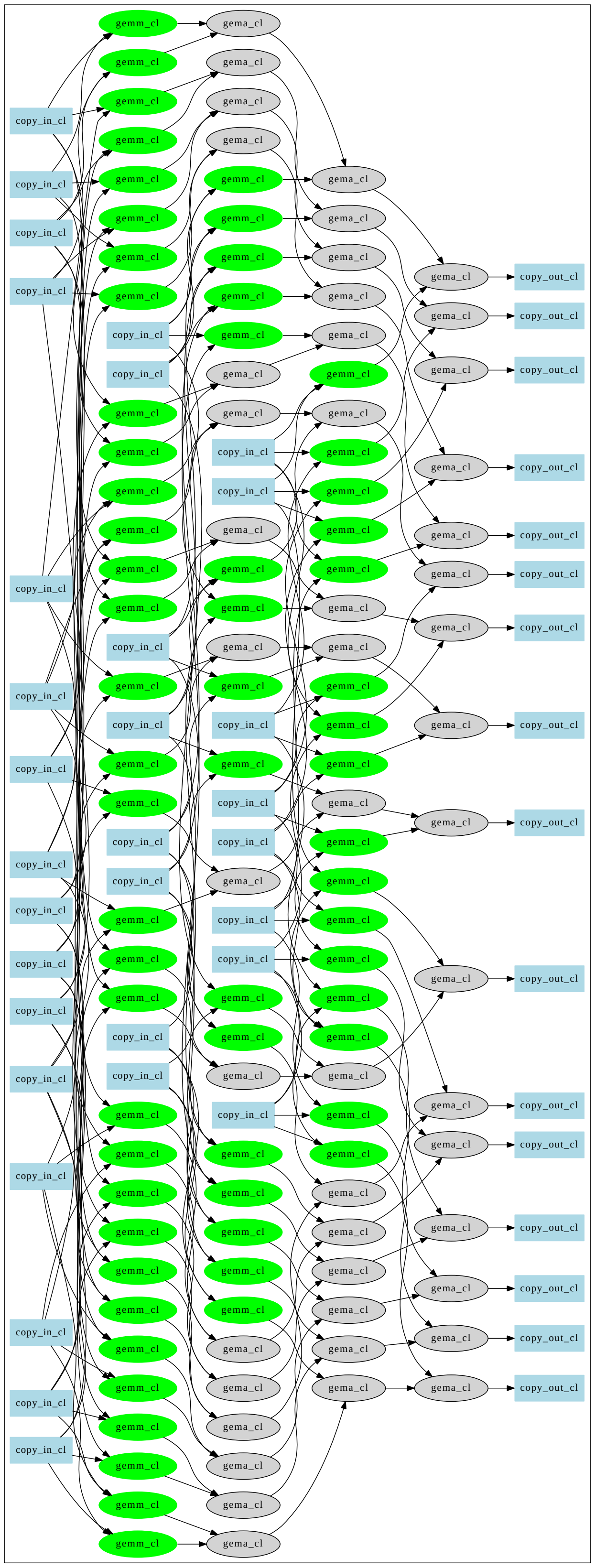


Figure B.4: Matrix Multiplication: 4 by 4 Tiling

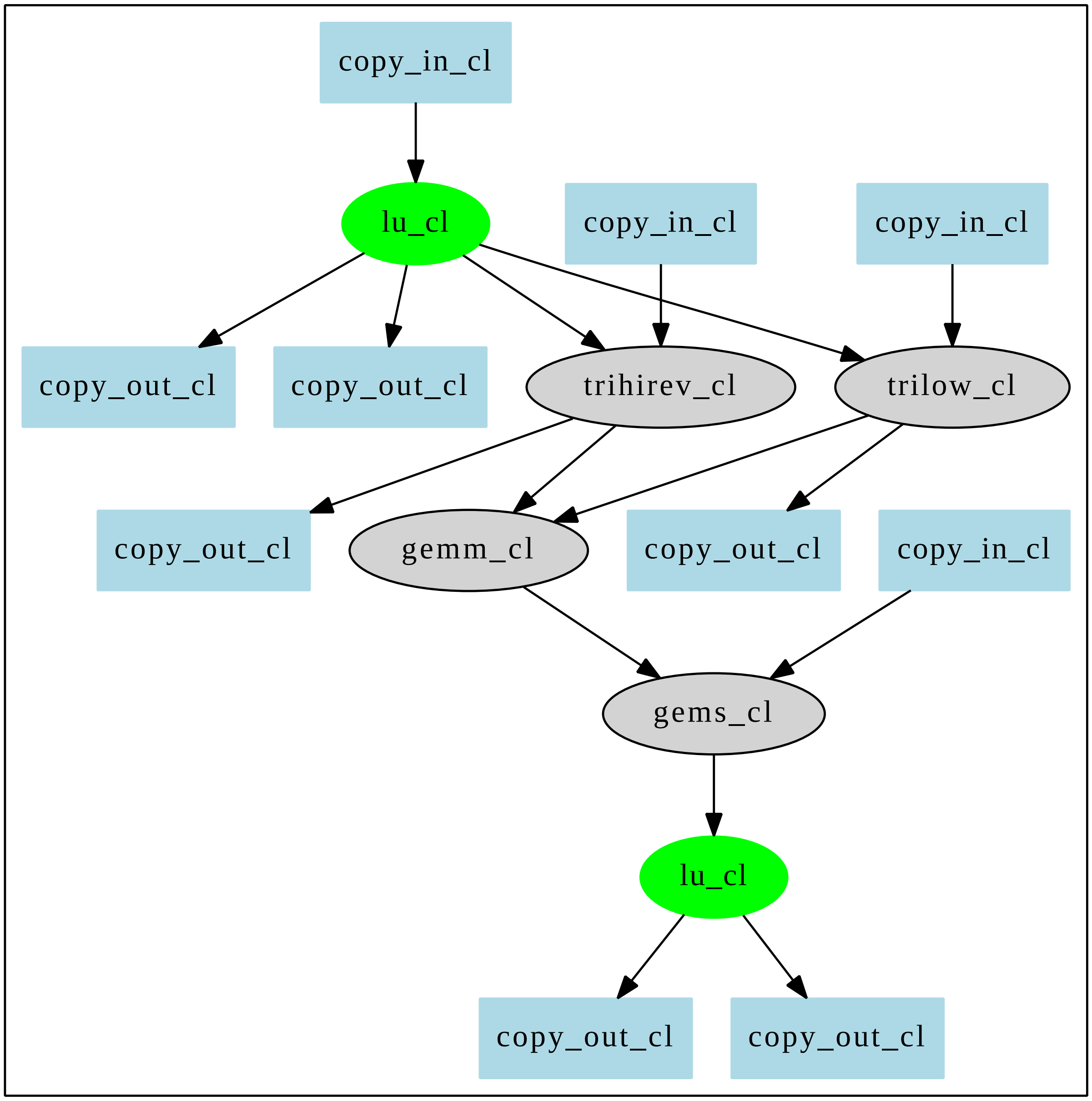


Figure B.5: LU Decomposition: 2 by 2 Tiling

Bibliography

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langu, H. Ltaeif, P. Luszcek, and S. Tomov. Numerical linear algebra on emerging architectures: the plasma and margma projects. *Journal of Physics: Conference Series*, 2009.

[2] J. Ansel and C. P. Chan. Petabricks. *ACM Crossroads*, 2010.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 2009.

[4] D. Barthou, S. Donadio, P. Carribault, A. X. Duchâteau, and W. Jalby. Loop optimization using hierarchical compilation and kernel decomposition. In *CGO*, 2007.

[5] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn. The Science of Deriving Dense Linear Algebra Algorithms. *ACM Trans. Math. Softw.*, Mar. 2005.

[6] J. Bilmes, K. Asonović, J. Demmel, D. Lam, and C. Chin. PHiPAC: A portable, high-performance, ANSI C coding methodology and its application to matrix multiply. LAPACK working note 111, University of Tennessee, 1996.

[7] J. C. Brodman, G. C. Evans, M. Manguoglu, A. Sameh, M. J. Garzarán, and D. Padua. A parallel numerical solver using hierarchically tiled arrays. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, LCPC'10, pages 46–61, 2011.

[8] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The impact of multicore on math software. In *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, PARA'06, pages 1–10, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] A. X. Duchateau. Automatic algorithm derivation and exploration in linear algebra for parallelism and locality, 2013. University of Illinois at Urbana-Champaign.

[10] D. Fabregat-Traver and P. Bientinesi. Knowledge-based automatic generation of partitioned matrix expressions. In *Proceedings of the 13th international conference on Computer algebra in scientific computing*, CASC'11, 2011.

[11] M. Frigo and S. G. Johnson. Fftw: An adaptive software architecture for the fft. In *Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, 1998.

[12] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. Dev.*, 41(6):737–756, Nov. 1997.

[13] U. o. T. Innovative Computing Laboratory. Plasma, 2012. http://icl.cs.utk.edu/plasma/pubs/index.html.

[14] Intel, 2012. Personnal communication.

[15] Intel®. Intel Math Kernel Library, 2011. http://software.intel.com/en-us/articles/intel-mkl/.

[16] Intel®. Intel Threading Building Blocks for C++, 2011. http://www.threadingbuildingblocks.org.

[17] Intel®. Intel Performance Primitives, 2012. http://software.intel.com/en-us/intel-ipp/.

[18] I. Jonsson and B. Kågström. Recursive blocked algorithms for solving triangular systems - Part I: one-sided and coupled Sylvester-type matrix equations. *ACM Trans. Math. Software*, 2002.

[19] I. Kodukula and K. Pingali. Data-centric transformations for locality enhancement. *Int. J. Parallel Program.*, 29(3):319–364, June 2001.

[20] MathWorks®. Matlab, 2013. http://www.mathworks.com/products/matlab/.

[21] E. Polizzi and A. H. Sameh. A parallel hybrid banded system solver: the {SPIKE} algorithm. *Parallel Computing*, 32(2):177 – 194, 2006. ¡ce:title¿Parallel Matrix Algorithms and Applications (PMAA04)¡/ce:title¿.

[22] 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"*, 2005.

[23] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for nonshared memory machines. In *In Supercomputing 91*, pages 111–120, 1991.

[24] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* O'Reilly, 1 edition, July 2007.

[25] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. *SIGPLAN Not.*, 38(5):77–90, May 2003.

[26] R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Sofware and the ATLAS Project. *Parallel Computing*, 2001.

[27] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, 1991.

[28] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas? *Proceedings of the IEEE*, feb. 2005.

[29] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 207–218, New York, NY, USA, 2012. ACM.