Hydra : Automatic Algorithm Exploration from Linear Algebra Equations

Alexandre X. Duchâteau
Department of Computer Science
University of Illinois
axdn@illinois.edu

David Padua
Department of Computer Science
University of Illinois
padua@illinois.edu

Denis Barthou
Labri
Université de Bordeaux
denis.barthou@inria.fr

Abstract

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 Intel's MKL.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization

General Terms Performance, Algorithms, Parallelism

Keywords Automatic Derivation, Linear Algebra

1. Introduction

Years of research have led to very powerful algorithms to solve linear algebra on all classes of machines. The algorithms and implementation strategies used for sequential systems differ from those used for parallel systems. For this reason, implementations that were developed for sequential machines may not be the ideal place to start when looking towards parallel solutions to a problem since some of the selections made to obtain efficient sequential codes may have to be changed in order to obtain as good parallel version.

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 ensuring 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 presents a simple triply nested loop that performs a matrix multiplication. Figure 2 is the same code, after application of a set of source optimization: tiling, scalar promotion, loop unrolling and loop interchange. The code now has 3 additional loops with larger strides, the innermost loop has two statements operating on scalars and single dimension arrays instead of three double dimension arrays. While these transformations may optimize sequential performance on a specific machine, they may hide parallelism to a parallelizing or vectorizing compiler or even from the programmer.

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 paper, we describe a system that automatically derive parallel codes from

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CGO'13 February 2013, Shenshen, China.

Copyright © 2013 ACM [to be supplied]. . . $10.00

```
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];
```

Figure 1. 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];
     }
    }
   }
  }
 }
}
```

Figure 2. Optimized Matrix Multiplication

high level descriptions of linear equations. This description includes the expressions in the mathematical equation, and information on the operands. Working from this equation, the system defines parameters to characterize a class of parallel solutions using a divide and conquer approach, then explores 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.

Outline. The rest of this paper is organized as follows. Section 2 describes the system. and Section 3 elaborates on the generator component that is at the core of our contribution. Section 4 presents results obtained on some matrix problems. Section 5 discusses related work and finally Section 6 describes our conclusions.

2. Overview of Hydra

Hydra is a code generator that starts from a mathematical description of matrix linear equations to solve, and generates parallel codes for multi-core architectures. The steps involved in this generation are described in Figure 3. The input is the description of the equation to solve and a collection of routines with their associated signatures. The signature identifies the form of the equation to be solved and the nature of the terms. For example, the signature

$$LT \cdot UNK = MT$$

represents an equation of the form $L \cdot X = M$ with X an unknown (denoted UNK) and L a lower triangular matrix (denoted LT). Relying on algebraic properties and on the shape of the matrices, Hydra applies a divide-and-conquer strategy to automatically generate different recurrent formulations of the initial problem. This crucial step is presented in the following section. As a result, multiple formulations are generated, they differ in terms of parallelism, computation grain and data locality.

To identify the fastest version produced by the generator, each one is executed on the actual target multi-core machine. This requires the target system to be available and some input data sets for the problem to solve. An alternative to this auto-tuning approach would be to rely on performance prediction [10] and keep for testing only the versions with the best performance prediction, or even remove the need for any execution. This possibility is not addressed in this paper. For input data sets, we assume that the user provides sample data sets or data generators. For dense linear algebra, the determining factor of performance is the size of the input data. It is thus important that the data provided matches the size or size range of the data with which the program would be used afterwards. The rest of this section describes the different components and their roles in more details.

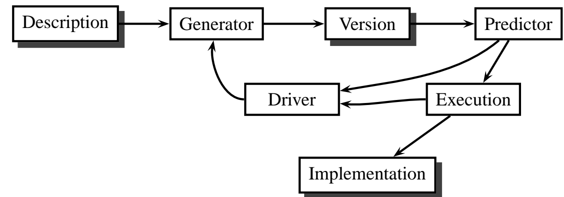

Figure 3. System graph

Our mathematical **description language** can represent matrix equations to solve. The only required information beyond the actual equation is the shapes of the matrices involved (e.g. triangular, symmetric matrices).

For a class of problems which can be said are *natively supported*, no additional information is required. In other words, it is not necessary to provide an algorithm to solve these problem. Figure 4 presents a full example with the description of a discrete triangular Sylvester equation (DTSY).

The input consists in:

- An equation on matrices. Basic matrix operations can be handled. So far, only addition, substraction, multiplication are supported in our current implementation. The matrices used in the equation are each described by their shapes, using the keywords `Square`, `Upper Triangular`, `Lower Triangular` and by their nature with the keyword `Unknown`. Matrices are assumed to be known by default, i.e. part of the input.
- The optional description of a library function, a kernel, that can be executed to solve this problem. Hydra can use this function for the base case of the recursion.
- An optional list of equations corresponding to other problems with the kernels to solve them; This list can be used by Hydra to solve subproblems of the initial problem.

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

%% Equation
A·X·B−X = C

%% Parameters
@name sylvester
@codelet sylsolv
@operands A B C
```

Figure 4. Discrete Triangular Sylvester Equation description

The main component is the **Generator**. The generator has a set of native transformation rules that it applies on equations. In particular, it transforms a single equation into a set of equations, in order to generate divide-and-conquer solutions to the problem. The results are task graphs that represent different possible implementations. The generator is further described in section 3.

In the future, a **Predictor** filter could be inserted between the generator and the empirical evaluation step. This component could increase the number of valid algorithms and implementation that can be evaluated in a fixed amount of time. Performance prediction is a difficult problem, but analysis of the generated task dependence graph can be performed to build bounds on achievable performance. The graphs width and the length of the critical path are examples of metrics that can be used to such end.

The **Code Generator / Execution** component performs empirical evaluation of the different versions. It first converts the task graph into code, using the StarPU[1] runtime scheduler API, compiles it and runs it on sample data to measure appropriate metrics (e.g. execution time, memory usage, power consumption, ...). Performance data are sent forth to the driver component. Sample data or data generators must be provided. Although in the current version, the output is selected based on a single data set, it is easy to extend the system so that it could generate input dependent libraries, that select a version as a function of characteristics of the input data, which in the case of dense linear algebra would be the size of the matrices and vectors.

If a codelet implementation is missing to translate a graph, a message is generated, presenting both the equation and its signature. Hydra can be used as an interactive development tool.

Finally, the **Driver** manages the process. In the simplest case, which is the one currently implemented, it restricts the search to a subset of all possible tilings of the equation, keeping track of the fastest version, applying successive recursive decompositions and stopping the generator once all cases have been tested. But it can also implement machine learning techniques to reduce the search space and improve filtering poor versions out.

3. Generator

The generator is Hydra's main component. It accepts the high level description of an equation, breaks it into multiple equations operating on smaller matrices, and possibly repeats this process by further breaking the generated equations. Along the way, the generators builds a task dependence graph specifying the necessary order in which these equations must be solved.

The generator operates on one equation at a time and on the dependence graph. At the beginning, the dependence graph has a single node representing the initial equation. At each step, the generator expands an equation by tiling its operands and then adjusting the dependence graph to incorporate the new equations and remove the one that was expanded.

Example 1: Consider the equation $M = L \cdot X$ with M a known matrix, L a known lower triangular matrix and X an unknown matrix. A way to expand this equation, is to convert each operand into a tiled array with two tiles along each dimension. Therefore, from

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

we obtain the following four equations.

$$M_{00} = L_{00} \cdot X_{00} \quad (1)$$
$$M_{01} = L_{00} \cdot X_{01} \quad (2)$$
$$M_{10} = L_{10} \cdot X_{00} + L_{11} \cdot X_{10} \quad (3)$$
$$M_{11} = L_{10} \cdot X_{01} + L_{11} \cdot X_{11} \quad (4)$$

The dependence graph associated with the initial $M = L \cdot X$ equation is a single node. For the equations resulting from the expansion the dependence graph has four nodes, one per equation. We label the nodes with the numbers given to each equation above. The graph has two arcs. One from node (1) to node (3) because X_{00} must be computed by solving equation (1) before equation (3) can be solved for X_{10} and another from node (2) to node (4) because X_{01} is needed to solve equation (4).

Example 2: Figure 5 illustrates the behavior of the generator for equation $L \cdot X \cdot U - X = C$. The first column of the the flow diagram, illustrates the initial steps followed by the generator. At the beginning, the original equation is available and the associated dependence graph is empty. Next, in a process we call *derivation*, the equation is expanded into four new equations while the dependence graph stays unchanged (i.e. empty). Finally, a process that we call *identification* generates a dependence graph linking the newly generated equations.

The last column of the figure, illustrates a step further down the road. An equation, the one labeled (4), has been selected for expansion. It is expanded into four new equations, and the identification step generates a dependence graph for

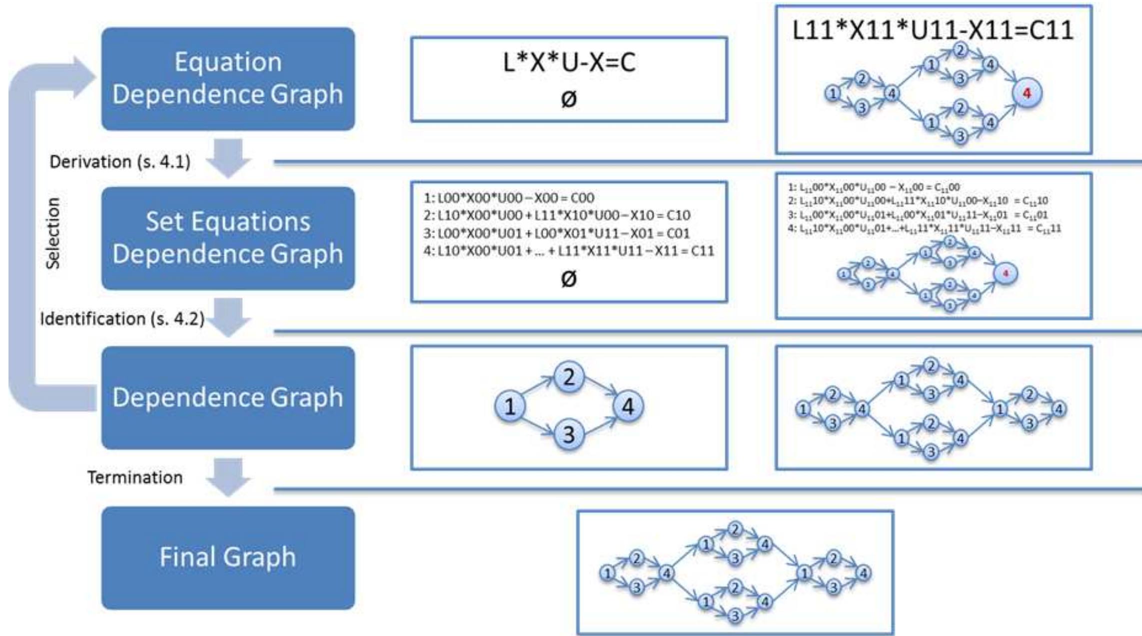

Figure 5. Generator overview with example

those new equations as well as integrates it into the larger dependence graph that represents the problem.

Termination of the generator can be decided by characteristics of the dependence graph, or by a recursion depth when all the operators have been tiled to the same granularity. In Example 2 we assume that termination happens after all equations have been expanded twice (recursion depth of two).

3.1 Equation Expansion or Derivation

Equation expansion is the process by which different algorithms are generated by the system. Different ways of tiling and different depths of recursion produce different algorithms. The first step of expansion, described in section 3.1.1, is to make sure that tiling is done in the right way. There would typically be numerous ways of tiling the operands and this defines the exploration space from where the final version of the solver will be selected. Section 3.1.2 describes how a solution is generated for one point in the exploration space.

3.1.1 Validity of tiling

The first step in the process of deriving an equation is to partition the operands of this equation into tiles. When considering matrix operations, one cannot arbitrarily partition the operands. Since we partition the operands in order to perform symbolic execution of the operation, a few basic rules must hold. e.g. when multiplying two matrices A and B, the number of columns of A must be equal to the number of rows of B. A blocking that does not conserve those rules is considered invalid and shouldn't be considered.

Instead of generating all possible tilings and then checking their validity, we ensure that only useful tilings are generated. To do so, we use the matrix operation properties to build a set of relations between the operand dimensions and only generate tilings satisfying those relations. Block and matrix sizes are at this point completely symbolic.

The shapes of the operands may also guide how blocking is applied to generate new equations with recognizable shapes. For example, we may want to block a triangular matrix so that it contains triangular matrices on the diagonal.

Figures **??** to **??** illustrate the first step of the process. Where we propagate the operands' dimensions. For this example, we look at equation $A \cdot X \cdot B - X = C$.

First (figure **??**) the system creates the operation tree assigning a tuple (x,y) to each operand where x and y are the number of blocks per column and row respectively (see figure 7). 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 and have no name in the original equation.

We then look at the virtual operands to assign them a tuple of dimensions. Knowing the dimensions of two operands of a matrix operation, we can deduce the dimensions of the result. i.e. the product of a $m \times n$-matrix by a $n \times l$-matrix will produce a $m \times l$-matrix. For example (figure **??**) we can assign the tuple (x_A, y_X) to the node that is the result of the product of A by X.

Now that all operands (real and virtual) have been assigned a set of dimensions, the system will examine each operational node in the tree and create the set of equations (1). For example, when looking at the node that represents the

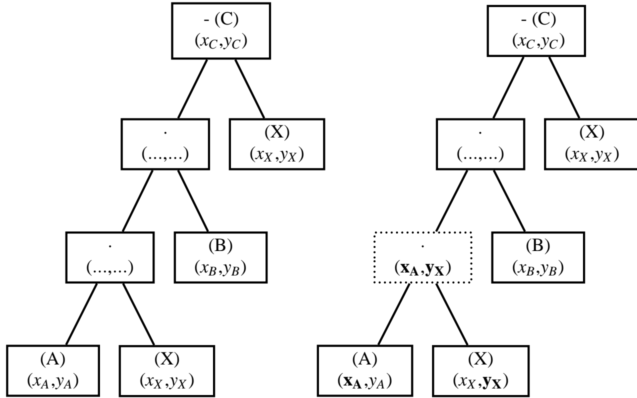

Figure 6. Operation Tree

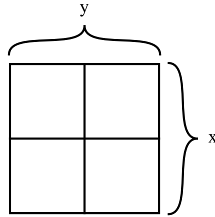

Figure 7. An operand's tiling dimensions

product of A.X by B. The number of columns of the left hand operand has to be equal to the number of lines of the right operand. That leads to equation $y_A = x_X$.

From equations $y_A = x_X$, $y_X = x_B$, $x_A = x_X$, $y_B = y_X$, $x_C = x_X$, and $y_C = y_X$ we get two sets of constraints on the number of tiles per dimension.

$$\begin{aligned} \{x_A = y_A &= x_X = x_C\} \\ \{x_B = y_B &= y_X = y_C\} \end{aligned} \quad (1)$$

Exploring all possible tilings of the problem is now reduced to finding two values and assigning them to their respective sets of variables.

3.1.2 Tiling

To derive the equation the system first partitions (tiles) its operands. It relies on the properties of matrix operations. Once the operands have been partitioned, we use symbolic execution of tiled operations to generate new sets of equations.

An important aspect of our system is how the operands' shapes are used to identify the unnecessary computation. 0-blocks (a matrix block that only contains 0 values) are absorbing elements for the matrix multiplication (i.e. $0 \cdot X = 0$) and identity elements for matrix addition (i.e. $0 + X = X$), thus computation involving 0-blocks can be simplified.

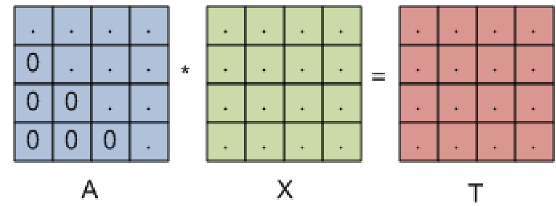

Figure 8. Original equation with operand shapes

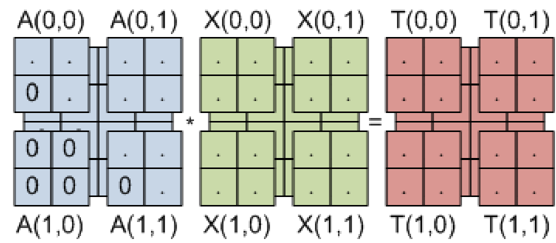

Figure 9. Tiling operands

```
T(0,0) = A(0,0)*X(0,0) + A(0,1)*X(1,0)
T(0,1) = A(0,0)*X(0,1) + A(0,1)*X(1,1)
T(1,0) = A(1,0)*X(0,0) + A(1,1)*X(1,0)
T(1,1) = A(1,0)*X(0,1) + A(1,1)*X(1,1)
```

Figure 10. Expansion generates new equations

```
T(0,0) = A(0,0)*X(0,0) + A(0,1)*X(1,0)
T(0,1) = A(0,0)*X(0,1) + A(0,1)*X(1,1)
T(1,0) =                 A(1,1)*X(1,0)
T(1,1) =                 A(1,1)*X(1,1)
```

Figure 11. Removing 0 operands

Figures 8 to 11 illustrates an example of such block partitioning. In 11 the bottom two equations have been simplified since it is known from the shape information that block A(1,0) is all zeros.

3.2 Identification and Dependence Graph Computation

Once a collection of new equations is created by tiling the operands (section 3.1), the generator proceeds to identify

the tasks described by those equations, and generate the dependence graph between those tasks. In this section, we describe how to achieve this.

3.2.1 Building Dependences

The crucial step in building the dependence graph is determining what is the input and output of each one of the equations. The input to Hydra identifies matrices whose values that are known at the outset. As discussed in Section 2, these are the matrices not annotated with the `Unknown` keyword in the input to the system. These matrices are placed by Hydra in the *input set* to the equations where they appear. Also all tiles of these known matrices and vectors are assumed to be input to the equations where they appear. All other operands are initially placed in the *output set* of the equations.

Example 3: Let us consider again the equation $M = L \cdot X$ from Example 1.

Because matrices M and L are known, we have that matrices L_{ij} and M_{ij} for $i, j \in \{0,1\}$ are also known, and because matrix X is unknown we have that matrices X_{ij} for $i, j \in \{0,1\}$ are unknown.

And here are a couple of input/output set examples :

Equation	Input set	Output set
$M_{00} = L_{00} \cdot X_{00}$	$\{M_{00}, L_{00}\}$	$\{X_{00}\}$
$M_{10} = L_{10} \cdot X_{00} + L_{11} \cdot X_{10}$	$\{M_{10}, L_{10}, L_{11}\}$	$\{X_{00}, X_{10}\}$

Algorithm 1 describes the process used to build the dependence graph for a newly created set of equations. The algorithm first (line 2) selects an *unidentified equation* from E. Then (line 3) it removes e from E converting in this way e into an *identified equation*. An equation e is selected if the system contains a kernel capable of solving the equation. This means that there is a kernel that accepts as input all matrices in the input set of the equation, solves the equation, and returns values for each of the matrices in the output set of the equation.

Algorithm 1 Building the dependence tree

1: **while** $E \neq \emptyset$ **do**
2: Select e in E % See section 3.2.2
3: $E \leftarrow E \setminus \{e\}$
4: **for all** $o \in e.output$ **do**
5: **for all** $d \in E$ **do**
6: **if** $o \in d.output$ **then**
7: $T \leftarrow T \cup \{(e; d)\}$
8: $d.output \leftarrow d.output \setminus \{o\}$
9: $d.input \leftarrow d.input \cup \{o\}$

All the matrices in the output set of e can, after identification, be considered as inputs to any of the equations in E. To reflect that fact, the loop on line 4 finds every equation in E that has e's output matrices in their own output set (until this point, those variables were unknown to every equation using them) and transfers it to their input set. In addition, a dependence edge is added between equation e and every equation that uses matrices from its output set. Section 3.2.2 discusses the details of this process of selection.

Once E is empty, every equation has been identified and added to the dependence graph. The process is then over.

Example 4: Let us consider one more time the equation

$$M = L \cdot X,$$

where M is a known matrix, L is a known lower triangular matrix and X is an unknown matrix. Each matrix is partitioned once along each dimension. The following equations are generated and added to set E with their associated input and output sets.

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

Equation (1) can clearly be solved using a triangular solver, but equation (3) cannot be solved until (1) has been solved because the value of $X_{0,0}$ is needed for its solution. Also, equation (2) can be solved, but equation (4) must wait for the solution to equation (2). When equation (1) is selected, the matrix X_{00} becomes a known variable. Since equation (3) has X_{00} in its output set, a dependence edge is created between (1) and (3) $T = T \cup \{((1) \rightarrow (3))\}$. And the input and output sets are updated. In addition, (1) is removed from E.

Other arcs in the dependence graph (those corresponding to incoming and outgoing arcs from the equation before expansion) can be trivially added since all that is needed is to connect elements in the output set in one equation to elements in the input set of other equations. The reason is that, except for newly expanded equations (which as mentioned above may have an incorrect number of matrices in the output set), the input and output set of all equations are properly defined.

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

3.2.2 Selection

The selection of an equation to add to the dependence tree is performed following algorithm 2.

This process consists in identifying which equations are solvable and thus can be added to the dependence graph.

First (line 1), we look for an equation that matches the original problem. This is done by direct comparison of the equation's signature to the signature of the original problem. Signatures are explained in detail in section 3.2.3.

If no such equation is found (line 4), we examine the equations looking for one that can be massaged into a match of the original problem. This is achieved by simplification of the equations signature, if an equation's signature can be made to match the main equation's signature then they

Algorithm 2 Equation Selection

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

are equivalent if some pre-processing computation is performed. For example, the equation $L \cdot X + B = M$ with L lower triangular and X unknown does not match the signature $LT \cdot UNK = MT$, but if we expand that into two equations: (1) $R = M - B$ and (2) $L \cdot X = M$ where (1) is a pre-processing step, we would get a match.

On line 7, the equation is replaced by a subgraph that contains the pre-processing steps and the new equation that matches the original. Simplification rules are explained in section 3.2.3 and the expansion step is explained in section 3.2.4.

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)

Example 5: In the first selection step in Example 4, both $M_{00} = L_{00} \cdot X_{00}$ and $M_{01} = L_{00} \cdot X_{01}$ are possible candidates. Both equations match the original problem : i.e. the product of a lower triangular matrix by an unknown equaled to a known matrix.

3.2.3 Signatures and Simplification

We define an equation's signature as the combination of the operations it contains and the shapes of its operands.

Let the following abbreviations stand:

- 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

For example, for LU decomposition ($L \cdot U = A$) the signature is :

$$UNK_LT \cdot UNK_UT = MT$$

The signature is used to identify the nodes. In particular to identify when the new generated equations are instances of the original problem on smaller data sets.

For the purpose of identification, simplification rules are defined on an equation's signature.

A few examples are :

- $MT + MT \Rightarrow MT$
- $MT \cdot MT \Rightarrow MT$
- $\ldots MT = MT \Rightarrow \ldots = MT - MT$

The rules presented have variants for each combination of shapes for the matrices. e.g.

- $LT \cdot LT \Rightarrow MT$
- $LT + LT \Rightarrow LT$

Example: Consider the equation $L \cdot X + X \cdot U = M$ with M a known matrix, L a known lower triangular matrix, U a known upper triangular matrix and X and unknown matrix. Each operand is blocked twice in each dimension.

The signature of the original problem is $LT \cdot UNK + UNK \cdot UT = MT$

Consider the derivated equation

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

at a stage where X_{01} is the only output. The equation signature is thus

$$LT \cdot UNK + MT \cdot MT + UNK \cdot UT = MT$$

and does not match the signature of the original problem.

$$\begin{aligned} & LT \cdot UNK + MT \cdot MT + UNK \cdot UT = MT \\ \Leftrightarrow \quad & LT \cdot UNK + MT + UNK \cdot UT = MT \\ \Leftrightarrow \quad & LT \cdot UNK + UNK \cdot UT = MT - MT \\ \Leftrightarrow \quad & LT \cdot UNK + UNK \cdot UT = MT \end{aligned}$$

After simplification, the signature matches the original problem.

3.2.4 Expansion

The expansion function allows to translate the simplifications applied on an equation's signature into tasks. Every simplification step applied on the signature is applied on the actual operands of the equation, generating a graph of simple solvable equations that lead to the new equation matching the original problem.

Example: Consider the equation and the simplification process described in the example in section 3.2.3.

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

For each simplification step that reduces the number of operands, the corresponding operation is added to the nodes equation set.

The set of operations corresponding to BLAS functions are built-in Hydra, that is able to match them automatically. In the previous example, $T_0 = X_{00} \cdot U_{01}$ thus matches a matrix multiply kernel.

4. Results

Starting from different linear problems on matrices and sequential kernels, Hydra generates automatically parallel codes solving these problems and resorting to these kernels. The task graph generated by Hydra is scheduled dynamically with StarPU runtime system [1].

In the following experiments, all sequential kernels used are from Intel MKL library [6]. In order to evaluate the capabilities of Hydra in terms of parallel code generation, we compare performance between the sequential MKL version with the best parallel version generated by Hydra for any given problem size. Besides, we compare the performance with the parallel MKL version. All our experiments were conducted on a 32-core (64 threads) platform composed of four 8-core Intel L7555 CPUs with 64GB of memory.

Figure 12 presents the results for the matrix multiplication. Here the decomposition obtained through Hydra corresponds to a block matrix multiplication. Note that these block matrix multiplications are not performed in-place, Hydra generates copies for each tile, improving here locality. We observe that the best parallel code generated by Hydra consistently outperforms the MKL parallel version of the matrix multiplication, for matrix sizes over 4000.

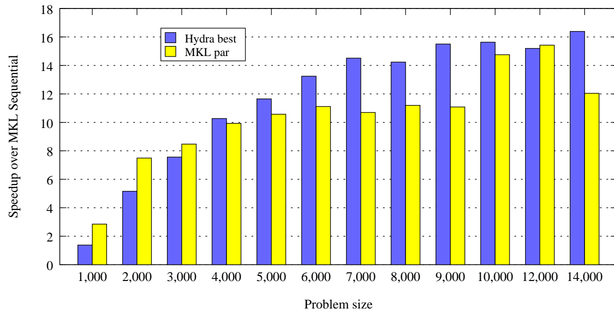

Figure 12. Matrix Matrix Multiplication: X = A*B

For the triangular solver, Figure 13 shows performance speed-ups compared to the sequential MKL and comparison with the parallel version. Performance of Hydra remains within roughly 10% of the parallel MKL performance. A more detailed analysis in Figure 14 shows the influence of the number of blocks on performance: tiling matrices in 10 by 10 blocks brings the best speed-up or large matrices.

Table 1 shows that for a blocking factor of 10, there are 1000 tasks created, the maximum number of tasks executable at the same time during the course of the execution is 90 (this is the width of the graph) and 255 copies of blocks

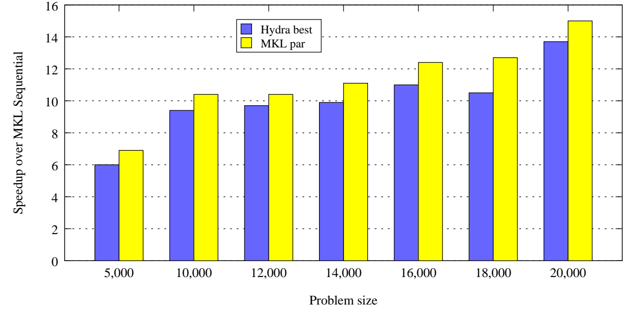

Figure 13. Triangular Solver : L*X = C

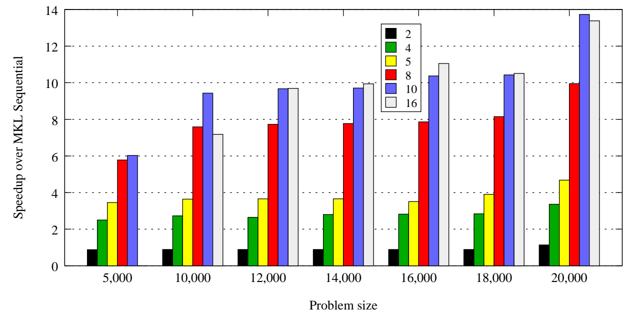

Figure 14. Exploring blocking factors for the triangular solver : L*X = C.

Blocking factor	Tasks	Max Parallelism	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 1. Triangular Solver: Characteristics of the different versions generated by Hydra, according to the blocking factor.

are performed. The high number of copies compared to the number of computational tasks may account for some performance loss.

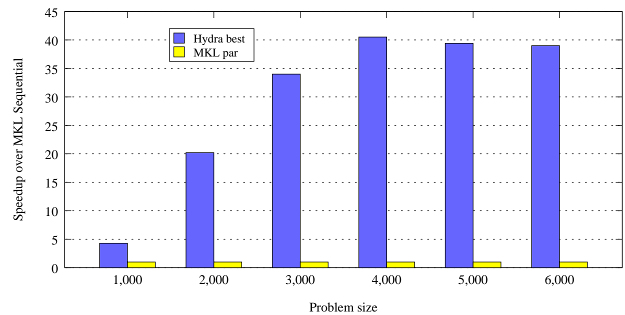

Figure 15. Triangular Sylvester: $AXB - X = C$

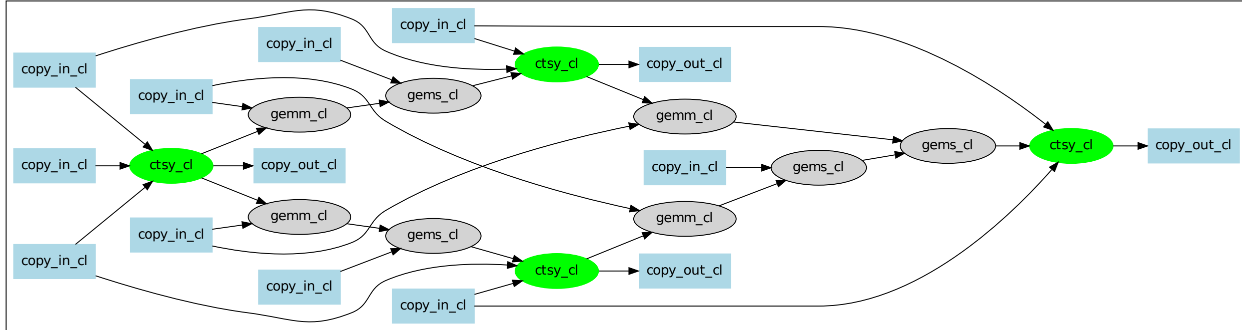

Figure 16. CTSY task graph for 2 by 2 blocking

Figure 15 show the speed-up of Hydra best code compared to the sequential MKL version. On a 32-core machine, the speed-up over 40 can be explained by the fact that Hydra decomposes the triangular Sylvester problem into sub-problems that have a higher sequential efficiency than MKL CTSY (such as matrix multiplication). Thus, the speed-up results from both the parallelization of the computation and from the use of efficient kernels. The task graph obtained for a 2 by 2 blocking of CTSY is shown in Figure 16. Square tasks are copy tasks, darker rounded tasks are smalled instances of CTSY and the others are various BLAS-3 operations. The method generated by Hydra to solve CTSY corresponds to the one described by Jonsson *et al.* [7].

Moreover, we observe that the parallel MKL version of CTSY has the same performance as the sequential one. This shows here all the benefits of Hydra: from sequential kernels and the initial formulation of the problem, we are able to generate automatically, with no efforts in manual code tuning, a parallel version of CTSY.

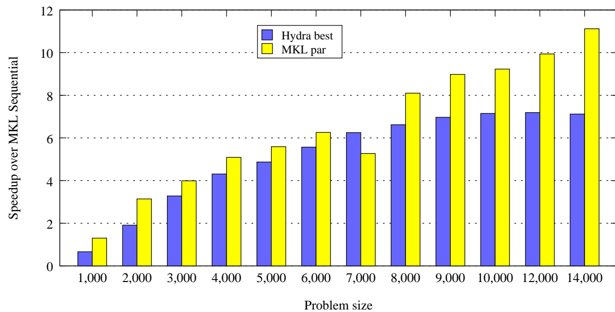

Figure 17. LU Factorization: L*U = A

Finally, Figure 17 shows performance for LU factorization. While the parallel MKL LU outperforms the code generated by Hydra, we note that the performance of Hydra consistently grows with the problem size.

5. Related Works

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.

ATLAS [9] 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.

The Spiral [8] 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, from which their automatic generation system can derive different algorithms and in the end implementations. They also use exhaustive search to evaluate performance and select the best implementation among all the versions generated by the system.

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

The Flame [3] project advocates goal-oriented programming. It offers a platform to develop algorithms in a systematical way to formally prove they achieve their goal. 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. 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 [4] 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.

Finally, work by Barthou *et al.* [2] on auto-tuning at source code level produce good results on matrix multipli-

cation, but suffered on more complex problems. 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 by 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.

6. Conclusion

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 no development effort: the user only needs to specify the equation to solve and provide sequential kernels. 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 for matrix multiplication and Sylvester triangular system resolution the parallel functions of Intel MKL library.

For future works, we plan to generalize Hydra for the generation of parallel codes for heterogeneous architectures. Indeed, one advantage of using a dynamic scheduler such as StarPU [1] 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 [6] and PLASMA [5] libraries). Moreover, Hydra offers the opportunity to generate parallel task graphs with non-uniform granularity, through different blocking sizes. Such graphs would then have coarser grain execution paths biased towards GPU execution and finer grain paths, better suited for multicore execution.

Acknowledgments

This work was funded by the Illinois-Intel Parallelism Center at the University of Illinois at Urbana-Champaign. The Center is sponsored by the Intel Corporation.

References

[1] 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*, 23:187–198, Feb. 2011. doi: 10.1002/cpe.1631. URL http://hal.inria.fr/inria-00550877.

[2] D. Barthou, S. Donadio, P. Carribault, A. X. Duchâteau, and W. Jalby. Loop optimization using hierarchical compilation and kernel decomposition. In *CGO*, pages 170–184, 2007.

[3] 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.*, 31(1):1–26, Mar. 2005.

[4] 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, pages 144–157, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23567-2. URL http://dl.acm.org/citation.cfm?id=2040148.2040160.

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

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

[7] 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*, 28(4):392–415, Dec. 2002. ISSN 0098-3500. doi: 10.1145/592843.592845. URL http://doi.acm.org/10.1145/592843.592845.

[8] 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"*, 93(2):232– 275, 2005.

[9] R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Sofware and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, 2001.

[10] 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*, 93(2):358 –386, feb. 2005. ISSN 0018-9219. doi: 10.1109/JPROC.2004.840444.