Hydra: Generation and Tuning of parallel solutions for linear algebra equations

Alexandre X. Duchâteau
University of Illinois at Urbana Champaign
Collaborators

• Thesis Advisors
  – Denis Barthou (Labri/INRIA Bordeaux)
  – David Padua (University of Illinois at UC)

• Future collaboration
  – starPU team, INRIA Bordeaux
INTRODUCTION

Objectives and Contributions
Objectives

- Compile linear algebra equations
  - Compute $X$ for $L \times X - X \times U = C$ [CTSY]
  - Compute $L$ and $U$ for $L \times U = A$ [LU]

- Generate efficient task parallel code
  - Identify tasks
  - Generate task dependence graph
Motivation

• Focus is on linear algebra
  – Start from high level description
  – No code or algorithm

• Derivation through blocking of operands
  – Data centric approach

• Derivation for parallelism
  – Output is a parallel task graph
Contributions

• A specification language
  – Express computation
  – Characterize operands (shapes)
  – Identify wanted result

• Derivation rules
  – Validity/applicability patterns
  – Operators symbolic execution rules
  – Dependence build engine
A detailed view of the generator

SYSTEM DESCRIPTION
Operands

- Status
  - Known, Unknown
- Shape
  - Triangular, diagonal
- Type
  - Matrix, (vector, scalar)
- Modifiers (transpose)
- (Sizes)
- (Density)

### All operands

• (Type inference)
Description language - Equation

• Simple equations
  – Assignments
    • $X = A \times B$

• Solvers
  – LU
  – Triangular Sylvester
  – $L \times X = B$
  – Cholesky

• Base for decomposition

%%% Operands
X: Unknown Matrix
L: Lower Triangular Square Matrix
U: Upper Triangular Square Matrix
C: Square Matrix

%%% Equation
$L \times X - X \times U = C$

%%% Parameters
@name ctsy
Description language - Parameters

• Drive code generation

• Set of parameters
  – Name
  – Codelet
  – Order of operands
  – Data type

• For customization
  – Default values used

%% Operands
X: Unknown Matrix
L: Lower Triangular Square Matrix
U: Upper Triangular Square Matrix
C: Square Matrix

%% Equation
L*X-X*U=C

%% Parameters
@name ctsy
Kernel Declaration

• Generate a full solution
  – Cost of full recursion
  – Usually not a good idea

• Use existing kernels and libraries
  – Already optimized
Kernel Performance

• Measure performance
  – Depend on size

• Guide exploration
  – Optimal nodes
  – Similar to ATLAS
Starpu Codelet

**Codelet declaration**

```c
void __gemm(void *buffers[], void *cl_arg);
struct starpu_codelet gemm_cl = {
    .where = STARPU_CPU,
    .cpu_funcs = {__gemm, NULL},
    .nbuffers = 3,
    .modes = {STARPU_R, STARPU_R, STARPU_RW}
};
```

**Kernel code**

```c
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]);
    // ...
    cblas_dgemm(CblasRowMajor,
                CBlasNoTrans, CBlasNoTrans,
                n, n, n,
                1.0,
                a, n, b, n,
                1.0,
                c, n);
}
```
Defining blocking space

• Blocking defines the space of solutions
  – Must only generate valid solutions

• Look at the equation’s operation tree
  – Each node gets a set of dimensions
  – Generate constraints depending on operation
Validity of Blocking

\[ \{ y_A = x_B \} \]
Valid Blockings - DTSY

Constraints

\[
\begin{align*}
y_A &= x_X \\
y_X &= x_B \\
x_A &= x_X \\
y_B &= y_X \\
x_C &= x_X \\
y_C &= y_X
\end{align*}
\]

\{ x_A, y_A, x_X, x_C \}
\{ x_B, y_B, y_X, y_C \}
Valid Blockings – DTSY (2)

$A^*X^*B - X = C \mid A \text{ lower triangular, } B \text{ upper triangular}$

- $xA = yA = xX = 2$
- $xB = yB = yX = 2$

- $xA = yA = xX = 2$
- $xB = yB = yX = 1$

- $xA = yA = xX = 3$
- $xB = yB = yX = 3$
Derivation example

\[
\begin{align*}
T(0,0) &= A(0,0) \times X(0,0) + A(0,1) \times X(1,0) \\
T(0,1) &= A(0,0) \times X(0,1) + A(0,1) \times X(1,1) \\
T(1,0) &= A(1,0) \times X(0,0) + A(1,1) \times X(1,0) \\
T(1,1) &= A(1,0) \times X(0,1) + A(1,1) \times X(1,1)
\end{align*}
\]

Sym. Exec. 


t = A \times X

blocking

Removal of 0-computation

\[
\begin{align*}
T(0,0) &= A(0,0) \times X(0,0) + A(0,1) \times X(1,0) \\
T(0,1) &= A(0,0) \times X(0,1) + A(0,1) \times X(1,1) \\
T(1,0) &= A(1,0) \times X(0,0) + A(1,1) \times X(1,0) \\
T(1,1) &= A(1,0) \times X(0,1) + A(1,1) \times X(1,1)
\end{align*}
\]
Operand characterization

- Blocks of X are outputs (unknown)
- A(0,0) and A(1,1) are lower triangular

<table>
<thead>
<tr>
<th>Equation</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(1,1) = A(1,1) \ast X(1,1)$</td>
<td>$T(1,1) A(1,1)$</td>
<td>$X(1,1)$</td>
</tr>
<tr>
<td>$T(1,0) = A(1,1) \ast X(1,0)$</td>
<td>$T(1,0) A(1,1)$</td>
<td>$X(1,0)$</td>
</tr>
<tr>
<td>$T(0,1) = A(0,0) \ast X(0,1) + A(0,1) \ast X(1,1)$</td>
<td>$T(0,1) A(0,0) A(0,1)$</td>
<td>$X(0,1) X(1,1)$</td>
</tr>
<tr>
<td>$T(0,0) = A(0,0) \ast X(0,0) + A(0,1) \ast X(1,0)$</td>
<td>$T(0,0) A(0,0) A(0,1)$</td>
<td>$X(0,0) X(1,0)$</td>
</tr>
</tbody>
</table>
Equation Signature

- Need to identify equations

- Identification through types and operators
  \[- A \times X = T : LT \times UNK = MT\]

- Set of simplification rules
  \[- UNK + \textcircled{MT} \times \textcircled{MT} \Rightarrow UNK + \textcircled{MT}\]
Identify task

- \( T(1,1) = A(1,1) \times X(1,1) \)
  - Signature: \( LT \times UNK = MT \)

- Instance of original problem
  - Solvable

- \( X(1,1) \) can now be considered known
Building dependence

• Find instances of $X(1,1)$
  – Shift in input set
  – Add dependence edge

<table>
<thead>
<tr>
<th>Equation</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(0,1) = A(0,0) \times X(0,1) + A(0,1) \times X(1,1)$</td>
<td>$T(0,1) \ A(0,0)$ $A(0,1) \ X(1,1)$</td>
<td>$X(0,1)$</td>
</tr>
</tbody>
</table>

New Dependence

$\{ T(1,1) = A(1,1) \times X(1,1) \rightarrow T(0,1) = A(0,0) \times X(0,1) + A(0,1) \times X(1,1) \}$
Signature Simplification

• \( T(0,1) = A(0,0) \times X(0,1) + A(0,1) \times X(1,1) \)
  1. \( MT = LT \times UNK + MT \times MT \)
  2. \( MT = LT \times UNK + MT \)
  3. \( MT - MT = LT \times UNK \)
  4. \( MT = LT \times UNK \)

• Match to the original problem!
## Post identification expansion

<table>
<thead>
<tr>
<th>Simplification step</th>
<th>Corresponding equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( MT = LT \ast UNK + MT \ast MT )</td>
<td>( T(0,1) = A(0,0) \ast X(0,1) + A(0,1) \ast X(1,1) )</td>
</tr>
<tr>
<td>( MT = LT \ast UNK + MT )</td>
<td>( T1 = A(01) \ast X(1,1) )</td>
</tr>
<tr>
<td></td>
<td>( T(0,1) = A(0,0) \ast X(0,1) + T1 )</td>
</tr>
<tr>
<td>( MT - MT = LT \ast UNK )</td>
<td></td>
</tr>
<tr>
<td>( MT = LT \ast UNK )</td>
<td>( T1 = A(01) \ast X(1,1) )</td>
</tr>
<tr>
<td></td>
<td>( T2 = T(0,1) - T1 )</td>
</tr>
<tr>
<td></td>
<td>( T2 = A(0,0) \ast X(0,1) )</td>
</tr>
</tbody>
</table>
Jointlab INRIA-UIUC
Simple graph example
Heterogeneous Graph Example
INITIAL RESULTS AND CHALLENGES
Challenges

• Version generation time
  – 5 minutes in generator
  – 20 minutes compiling

• Generated code size
  – 500k lines of code in single function
  – (icc segfaults)
Trisolve \((L^*X=B)\)
Conclusion

• Faster development cycle for architectures
  – No time to hand-tune everything anymore
  – Can’t hand tune for every HW iteration

• Increased complexity
  – Heterogeneous systems

• Description of an automatic methodology
  – Leverage existing “small scale” libraries
AU CAS OU
Exploiting heterogeneous machines: starPU Runtime

- **Goal**: Scheduling (≠ offloading) tasks over heterogeneous machines
  - CPU + GPU + SPU = *PU
  - Auto-tuning of performance models
  - Optimization of memory transfers

- **Target for**
  - Compilers
    - StarSs [UPC], HMPP [CAPS]
  - Libraries
    - PLASMA/MAGMA [UTK]
  - Applications
    - Fast multipole methods [CESTA]

- **StarPU provides an Open Scheduling platform**
  - Scheduling algorithm = plugins
Overview of StarPU
Maximizing PU occupancy, minimizing data transfers

• Principle
  – Accept tasks that may have multiple implementations
    • Together with potential interdependencies
      – Leads to a dynamic acyclic graph of tasks
      – Data-flow approach
  – Provide a high-level data management layer
    • Application should only describe
      – Which data may be accessed by tasks
      – How data may be divided
Code styles and optimization

```c
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];
```

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

• Generation of multiple solutions
  – Algorithmic exploration

• Theoretical evaluation
  – Usually unreliable
  – Filtering heuristic

• Empirical evaluation (execution)
  – slow (very slow when looking at bad versions)
Generalized system overview

- **Predictor**
  - Filters what versions are actually executed
- **Driver**
  - Guide the search