Compiler-Assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors

Muthu Baskaran¹  Naga Vydyananthan¹
Uday Bondhugula¹  J. (Ram) Ramanujam²
Atanas Rountev¹  P. Sadayappan¹

¹The Ohio State University
²Louisiana State University

Funded by NSF
Automatic parallelization of regular scientific programs on multi-core systems using polyhedral frameworks

Case for dynamic scheduling – (Dongarra’s PLASMA group work)
- Excessive barrier synchronization
- Load imbalance – varying degrees of parallelism

Support from compile-time and run-time systems required for parallel application development

Idea: Develop a fully-automatic approach for asynchronous load-balanced parallel execution
- To automatically generate tiled code along with additional helper code at compile time
- The helper code at run time extracts inter-tile data dependences and schedules the tiles

Achieved significant improvement over programs automatically parallelized using affine frameworks
Static (Affine) Scheduling
Polyhedral Model

- Effective use of Polyhedral Compiler Techniques
  - Powerful for optimizing regular scientific programs
  - Automatic parallelization and data locality optimization

- Polyhedral Model
  - An algebraic framework for representing affine programs statement domains, dependences, array access functions – and affine program transformations

- Affine programs
  - Loop bounds and array accesses - affine functions of outer loop variables, constants and program parameters
for (i=1; i<=7; i++)
   for (j=2; j<=6; j++)
      \[ S_1: a[i][j] = a[j][i] + a[i][j-1]; \]

\[ F_{3a}(\mathbf{x}_{S_1}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} \]

\[ D_{S_1}(\mathbf{x}_{S_1}) = \begin{bmatrix} 1 & 0 & -1 \\ -1 & 0 & 7 \\ 0 & 1 & -2 \\ 0 & -1 & 6 \end{bmatrix} \cdot \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} \geq 0 \]
for (i=1; i<=7; i++)
for (j=2; j<=6; j++)
S1: a[i][j] = a[j][i] + a[i][j-1];

Dependence Polytope
An instance of statement t (i_t) depends on an instance of statement s (i_s)

- i_s is a valid point in D_s
- i_t is a valid point in D_t
- i_s executed before i_t
- Access same memory location
- h-transform: relates target instance to source instance involved in last conflicting access

\[
\begin{pmatrix}
D_s & 0 \\
0 & D_t \\
\end{pmatrix}
\begin{pmatrix}
i_s \\
i_t \\
\end{pmatrix}
\begin{pmatrix}
\geq 0 \\
= 0 \\
\end{pmatrix}
\]
Affine Transformations

- Loop transformations defined using affine mapping functions
  - Original iteration space => Transformed iteration space
  - A one-dimensional affine transform ($\Phi$) for a statement $S$ is given by
    \[
    \Phi_S(x_S) = C_S \cdot \begin{pmatrix} x_S \\ n \\ 1 \end{pmatrix}
    \]
  - $\Phi$ represents a new loop in the transformed space
  - Set of linearly independent affine transforms
    - Define the transformed space
    - Define tiling hyperplanes for tiled code generation
TILING

For $i=0$ to $N-1$ do:
$x[i] = 0$;

For $j=0$ to $N-1$ do:
$S: x[i] += a[j][i] \cdot y[j]$;

For $it = 0$ to $\text{floor}(N-1,32)$ do:
For $jt = 0$ to $\text{floor}(N-1,32)$ do:
$\ldots$

For $i = \text{max}(32it,0)$ to $\text{min}(32it+31,N-1)$ do:
For $j = \text{max}(32jt,1)$ to $\text{min}(32jt+31,N-1)$ do:
$S: x[i] += a[j][i] \cdot y[j]$;

Tiled iteration space
- Higher-dimensional polytope
- Supernode iterators
- Intra-tile iterators
Parallel Tiled Code Generation

- Perform tiling
  - Create tile loops and element (intra-tile) loops
- Extract (coarse-grained) parallelism in tiled code
  - Identify synchronization-free “tile” or “supernode” loops
    - Find all $\Phi$ that have no dependence along it
Parallel Tiled Code Generation

- Exploit pipeline parallelism in the tile space
  - Find all $\Phi$ that carry a dependence (in the forward direction), say $(\Phi_1, \Phi_2, \ldots, \Phi_n)$
  - $\Phi_1 + \Phi_2 + \ldots + \Phi_n$ carries all dependences
  - Make it the outer sequential tile loop that represents a wavefront of parallel tiles
  - Make $\Phi_1, \Phi_2, \ldots, \Phi_n-1$ as inner parallel loops
PLUTO

- State-of-the-art polyhedral model based automatic parallelization system
- First approach to explicitly model tiling in a polyhedral transformation framework
- Finds a set of “good” affine transforms or tiling hyperplanes to address two key issues
  - effective extraction of coarse-grained parallelism
  - data locality optimization
- Handles imperfectly nested loops
- Uses state-of-the-art code generator CLooG
  - Takes original statement domains and affine transforms (which define tiling hyperplanes) to generate transformed code
Affine Compiler Frameworks

- **Pros**
  - Powerful algebraic framework for abstracting dependences and transformations
  - Enables the feasibility of automatic parallelization
    - Eases the burden of programmers

- **Cons**
  - Generated parallel code may have excessive barrier synchronization due to affine schedules
  - Loss of efficiency on multi-core systems due to load imbalance and poor scalability!
Aim and Approach (1)

- Can we develop an automatic parallelization approach for asynchronous, load-balanced parallel execution?

- Utilize the powerful polyhedral model
  - To generate tiling hyperplanes (generate tiled code)
  - To derive inter-tile dependences

- Effectively schedule the tiles for parallel execution on the processor cores of a multi-core system
  - Dynamic (run-time) scheduling
Aim and Approach (2)

- Each tile identified by our affine framework is a “task” that is scheduled for execution
- Compile-time generation of the following code segments
  - Code executed within a tile or task
  - Code to extract inter-tile (inter-task) dependences in the form of task dependence graph (TDG)
  - Code to dynamically schedule the tasks using critical path analysis on TDG to prioritize tasks

Run-time execution of the compile-time generated code for efficient asynchronous parallelism
System Overview

Pluto Optimization System

Task Dependence Graph Generator

Task Scheduler

Input code → Tiled code → Meta-info (Dependence, Transformation) → TDG gen code

TDG Code Generator

Output code (TDG, Task, Scheduling code)

Inter-tile (Task) dependences

Meta-info (Dependence, Transformation)

Tiled code

Task Dependence Graph Generator

Parallel Task code

System Overview
Task Dependence Graph Generation

- Task Dependence Graph (TDG)
  - DAG
    - Vertices – Tiles or tasks
    - Edges – Dependence between the corresponding tasks
  - Vertices and edges may be assigned weights
    - Vertex weight – based on task execution
    - Edge weight – based on communication between tasks
    - Current implementation: Unit weights for vertices and zero weights for edges
Task Dependence Graph Generation

- Compile-time generation of TDG generation code
  - Code to enumerate the vertices (or tasks)
    - Scan the iteration space polytopes of all statements in the tiled domain, projected to contain only the supernode iterators (*tile loop indices*)
    - CLooG loop generator is used for scanning the polytopes
  - Code to create the edges (or dependences among tasks)
    - Requires extraction of inter-tile dependences
    - This also done using CLooG
Inter-tile Dependence Abstraction
Static (Affine) Scheduling
Dynamic Scheduling

- Scheduling strategy: *critical path analysis* for prioritizing tasks in TDG
- Priority metrics associated with vertices
  - $topL(v)$ - length of the longest path from the source vertex (i.e., the vertex with no predecessors) in $G$ to $v$, excluding the vertex weight of $v$
  - $bottomL(v)$ - length of the longest path from $v$ to the sink (vertex with no children), including the vertex weight of $v$
- Tasks are prioritized based on
  - sum of their top and bottom levels; or
  - just the bottom level
Tasks are scheduled for execution based on:
- completion of predecessor tasks
- \( \text{bottomL}(v) \) priority
- availability of processor core
Affine vs. Dynamic Scheduling
Experiments

Experimental Setup

- a quad-core Intel Core 2 Quad Q6600 CPU
  - clocked at 2.4 GHz (1066 MHz FSB)
  - 8MB L2 cache (4MB shared per core pair)
  - Linux kernel version 2.6.22 (x86-64)
- a dual quad core Intel Xeon(R) E5345 CPU
  - clocked at 2.33 GHz
  - each chip having a 8MB L2 cache (4MB shared per core pair)
  - Linux kernel version 2.6.18
- ICC 10.x compiler
  - Options: -fast –funroll-loops (-openmp for parallelized code)
Experiments

Performance of LU on Quad Core

GFLOPs

Number of Cores

0 1 2 3 4 5 6 7 8 9

1 2 3 4

Statically scheduled LU

Dynamically scheduled LU

- Statically scheduled LU
- Dynamically scheduled LU
Experiments

Speedup of LU on Quad Core

Number of cores

Speedup

Statically scheduled LU
Dynamically scheduled LU
Experiments

Performance of LU on Dual Quad Core

GFLOPs vs. Number of cores

- Statically scheduled LU
- Dynamically scheduled LU
Experiments

Speedup of LU on Dual Quad Core

Number of cores vs. Speedup for Statically and Dynamically scheduled LU operations.

- Statically scheduled LU
- Dynamically scheduled LU

Graph shows a linear increase in speedup with an increase in the number of cores.
Experiments

Performance of Cholesky on Quad Core

- **Experiments**

- **Performance of Cholesky on Quad Core**

  - **Number of Cores**

    - 1
    - 2
    - 3
    - 4

  - **Speedup**

    - Pluto generated Cholesky
    - Dynamically scheduled Pluto generated
Experiments

Performance of Cholesky on Dual Quad Core

![Graph showing the performance of Cholesky on Dual Quad Core. The graph plots speedup against the number of cores. There are two lines: one for Pluto generated Cholesky and another for dynamically scheduled Pluto generated Cholesky. The speedup increases linearly with the number of cores.](image-url)
Discussion - 1

- Absolute achieved GFLOPs performance is currently less than the machine peak by over a factor of 2
- Single-node performance sub-optimal
  - Poor vectorization
  - No pre-optimized tuned kernels
    - e.g. BLAS kernels like DGEMM in LU code
- Work in progress to provide
  - Compile-time tile DAG generation (fixed loop limits, tile sizes, parameterized DAGs)
  - Compile-time tile DAG scheduling (fixed loop limits, fixed tile sizes)
  - Improved vectorization capability
  - Identification of tiles where pre-optimized kernels can be substituted
Discussion - 2

- Can we use other frameworks in this context?
  Yes, we can use this framework to generate “tiles” for the Concurrent Collections work (Kathy Knobe’s talk from earlier today)
    - Can benefit from flexible scheduling in CnC
    - Can result in lower overhead

- Search over tile sizes and unroll factors can be done with Orio (a syntactic transformer based on annotations; developed with ANL)
Related Work

- Dongarra et al. - PLASMA (Parallel Linear Algebra for Scalable Multi-core Architectures)
  - LAPACK codes optimization - Manual rewriting of LAPACK routines
  - Run-time scheduling framework
- Robert van de Geijn et al. - FLAME
- Dynamic run-time parallelization [LRPD, Mitosis, etc.]
  - Basic difference: Dynamic scheduling of loop computations amenable to compile-time characterization of dependences
- A whole lot of work on DAG scheduling
Summary

- Developed a fully-automatic approach for asynchronous load balanced parallel execution on multi-core systems

- Basic idea
  - To automatically generate tiled code along with additional helper code at compile time
  - Role of helper code at run time
    - to dynamically extract inter-tile data dependences
    - to dynamically schedule the tiles on the processor cores

- Achieved significant improvement over programs automatically parallelized using affine compiler frameworks

- Another dimension to auto-tuning: exploring static vs. dynamic scheduling