Tiling: Progress and Challenges

P. (Saday) Sadayappan
Ohio State University

SIAM PP 14 Tiling MiniSymposium
Tiling: Progress

- Michael Wolfe’s compiler challenge in the early 90’s: Automatically generate all 6 “ijk” forms for Cholesky factorization
- Polyhedral compiler frameworks can transform and generate tiled code for imperfectly nested affine codes
- Advances in model-driven and auto-tuning based tile size optimization
- Tiled parallel execution of stencil computations
  - Standard polyhedral tiling (skew + rectangular tiles) causes loss of inter-tile parallelism along skew direction
  - Several recently proposed tiling approaches: overlapped tiling, split tiling, diamond tiling, hexagonal tiling
Tiling: Questions

• How can the research community translate advances to real benefits on a range of real application codes?

• How can we assess how close to “optimal” a tiled code is?
  – Is there potential for further data locality enhancement, or are we hitting inherent “lower bounds” for the computation?
Dynamic Analysis for Characterization of Data Locality Potential*

INRIA Compsys/ ENS Lyon
Fabrice Rastello

Louisiana State University
J. (Ram) Ramanujam

UCLA
Louis-Noel Pouchet

Ohio State University
Naznin Fauzia,
Venmugil Elango,
Mahesh Ravishankar,
Nasko Rountev,
P. (Saday) Sadayappan

*Supported by the U.S. National Science Foundation & Dept. of Energy
Motivation

• Data movement is much more expensive in computer systems than arithmetic operations (Flops)
  – Performance: latency as well as throughput
  – Energy
• Computational complexity alone (number of ops executed) can no longer be sole (or even primary) criterion of algorithm choice
• But what is the inherent data movement complexity of an alg.?  
  – Computational complexity well understood; invariant to transforms
  – Data access complexity is not well characterized today: cost is affected by code transformations and also capacity of registers/caches
• Understanding data movement complexity is important:
  – Assessing manual/compiler optimizations: How much further improvement potential is there, beyond current optimizations?
  – Algorithm choice between alternatives – e.g., will conjugate gradient solver or GMRES solver be better in the future?
  – Arch. parameters: minimum cache capacity and/or bus bw. needed to support inherent data movement needs of an alg.
Counting Data Movement versus Computation

- 2D Seidel code has computational complexity of $(N-1)^2$ operations
- Tiled version of Seidel code also has exactly the same computational complexity
- But the number of cache misses (data movements from memory) for the two are different
  - Also depends on cache capacity
Computational DAG Abstraction

- Untiled and tiled versions of 1-sweep 2D Seidel code have same op-count but different cache miss profiles.

- Given some code, how can we reason about data access costs for all possible equivalent transformed versions of that code?

- The Computational DAG is a common abstraction for all valid transformed versions.

- CDAG abstracts away:
  - Particular sequential schedule of operations
  - Binding of data values to memory locations

```plaintext
for (i=1; i<N-1; i++)
  for (j=1; j<N-1; j++)
```

```plaintext
for (it = 1; it<N-1; it += 2)
  for (jt = 1; jt<N-1; jt += 2)
    for (i = it; i < min(it+B, N-1); i++)
      for (j = jt; j < min(jt+B, N-1); j++)
```
Data Access Cost of a CDAG

- Given a CDAG, what is the minimum possible data movement to execute it in a system with a 2-level memory hierarchy?

- Develop upper bound on min-cost
- Minimum possible data access cost?
- Develop lower bound on min-cost
Data Access Cost: Cache Hit/Miss

For (i=1; i<N-1; i++)
for (j=1; j<N-1; j++)

For (it = 1; it<N-1; it += B)
for (jt = 1; jt<N-1; jt += B)
for (i = it; i < min(it+B, N-1); i++)
for (j = jt; j < min(jt+B, N-1); j++)

2D-Seidel, N = 200, Tile Size = 25
Data Access Cost: Cache Hit/Miss

for (i=1; i<N-1; i++)
    for (j=1; j<N-1; j++)

for (it = 1; it<N-1; it += B)
    for (jt = 1; jt<N-1; jt += B)
        for (i = it; i < min(it+B, N-1); i++)
            for (j = jt; j < min(jt+B, N-1); j++)

2D-Seidel: N = 200, Tile Size = 25

Miss Count vs Machine Cache Size (Bytes)
Untiled - Tiled

Hit Count vs Machine Cache Size (Bytes)

2D-Seidel: N = 200, Tile Size = 25
Data Access Cost: Bytes/Flop

```
for (i=1; i<N-1; i++)
for (j=1; j<N-1; j++)
```

```
for (it = 1; it < N-1; it += B)
for (jt = 1; jt < N-1; jt += B)
    for (i = it; i < min(it+B, N-1); i++)
        for (j = jt; j < min(jt+B, N-1); j++)
```

2D-Seidel : N = 200, Tile Size = 25

![Graph showing data access cost comparison between untiled and tiled methods.](image)
Assessing Potential for Improvement

- Find a different schedule of operations
  - Instrument code to generate trace of instructions and register/memory addresses
  - A posteriori, generate CDAG for computation
  - Perform convex partitioning of CDAG (irregular tiling)
  - Reordered schedule based on “tiled” execution of CDAG

- Provides insights but not automated code transformation
  - Identify promising code regions for manual transformation
  - Assess the effectiveness of compiler transformations
Tiling: Convex Partitioning of CDAG

- Perform **convex partitioning** of the CDAG
- Assign each node in a single convex component
- Convex partitioning of a CDAG ≈ Tiling the iteration space
- Each component is convex
  - Can be executed atomically
  - No cyclic data dependence among components
- Topologically sorted order of the convex components
  => a valid execution order
Tiling: Convex Partitioning of CDAG

- Perform *convex partitioning* of the CDAG
- Assign each node in a single convex component
- Convex partitioning of a CDAG ≈ Tiling the iteration space
- Each component is convex
  - Can be executed atomically
  - No cyclic data dependence among components
- Topologically sorted order of the convex components
  => a valid execution order
Tiling: Convex Partitioning of CDAG

- Perform **convex partitioning** of the CDAG
- Assign each node in a single convex component
- Convex partitioning of a CDAG \( \approx \) Tiling the iteration space
- Each component is convex
  - Can be executed atomically
  - No cyclic data dependence among components
- Topologically sorted order of the convex components
  => a valid execution order
Convex Partitioning Heuristic

• Inputs:
  – CDAG
  – Cache size C
  – Priority defining parameters: depth vs. breadth

• Output: a modified execution order

• Convex partitioning: convex-component growing heuristic
  – Successively adds ready vertices into the component
  – Keep growing the convex component until a capacity constraint is exceeded
Case Studies

• Givens Rotation
  – Dynamic CDAG Analysis showed potential for improvement
  – Code was manually modified
  – Modified code could be tiled by Pluto tool (original could not)

• Householder transformation
  – Dynamic analysis did not show potential for improvement

• Floyd Warshall all-pairs shortest paths algorithm
  – Code in basic form not 3D tileable due to dependences
  – Dynamic CDAG Analysis showed potential for improvement
  – This was a surprising result; closer re-examination of code showed it could actually be tiled after manual index set splitting
Case studies

Givens: Maxlive=100, Varying Priorities

- Analysis showed potential
- Manually modified to apply automatic tiling tool
Givens: Configuration = Multi:Depth, Maxlive 100

- Original
- Convex-partitioning
- Tiled

Bytes/FLOP vs. Machine Cache Size (Bytes)
Case studies

Householder: Maxlive=100, Varying Priorities

![Graph showing machine cache size vs. bytes/FLOP for different priority schemes.](image)
Case studies

Householder: Config. = Multi: Breadth, Varying Maxlive
Summary

- New approach to estimating potential for enhancement of data locality for a given code
  - Dynamic generation of CDAG
  - Reorder operations via convex partitioning of CDAG
- Case studies illustrate use of dynamic analysis
- Potential Uses
  - Improve performance of code (via manual restructuring)
  - Assess effectiveness of compiler optimizations
  - Estimate machine parameters for future technologies
- Future Work
  - Address trace size limitations: out-of-core analysis
  - Improved quality convex partitioning heuristics
  - Make the tool more user-friendly