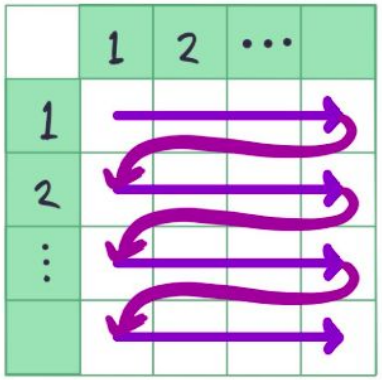
A Practical Tile Size Selection Model for Affine Loop Nests

Authors: Kumudha Narasimhan, Aravind Acharya, Abhinav Baid, Uday Bondhugula

Presented by Group 29:
Divya Shekar, Sashwat Mahalingam, Srikrishnan Ravichandran, Nikhil Janarthan Surendran, Nicholas Mellon

Loop Tiling

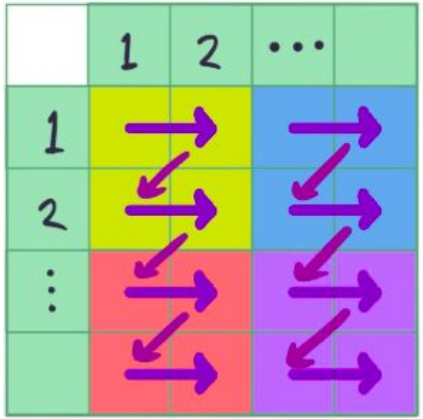
- Optimization technique for loop nests
- Partition large loop iteration space into tiles
  - Data remains in cache longer
  - Cache-friendly chunks



Loop without tiling

Benefits

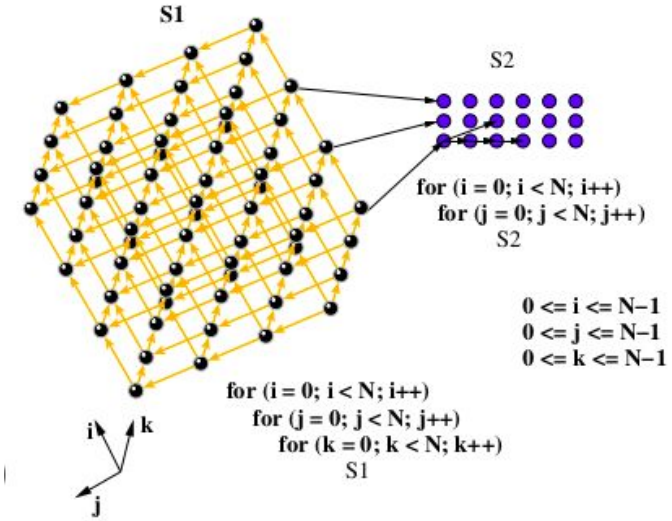
- Improved data locality
- More efficient cache re-use
- Increased memory subsystem performance



Loop with tiling

Vectorization

- Code transformation to process multiple data elements simultaneously
- **Single Instruction Multiple Data (SIMD)**
- Encode iteration spaces as affine expressions via polyhedral compilation



Polyhedral Compilers

- Represent loop nests as mathematical polyhedra
- Systematic analysis and loop transformation

Motivation

- Memory hierarchy and cache characteristics vary across compilers
  - “One-size-fits-all” tile sizes suboptimal
- Smaller tile sizes - Underutilization of cache
- Larger tile sizes - Cache misses
- Experimental tile size tuning is costly
  - High computation overhead
  - Time consuming
- Need for adaptable tile size selection for affine access patterns

Problem Statement

- Generic and fast model for tile size selection in affine loop nests
  - Suitable for adoption in general-purpose compiler infrastructure like MLIR

- Effective utilization of cache while considering vectorization, prefetching and load balancing

- To find tile sizes for n dimensions of a loop nest, find zeros of the polynomial of degree n

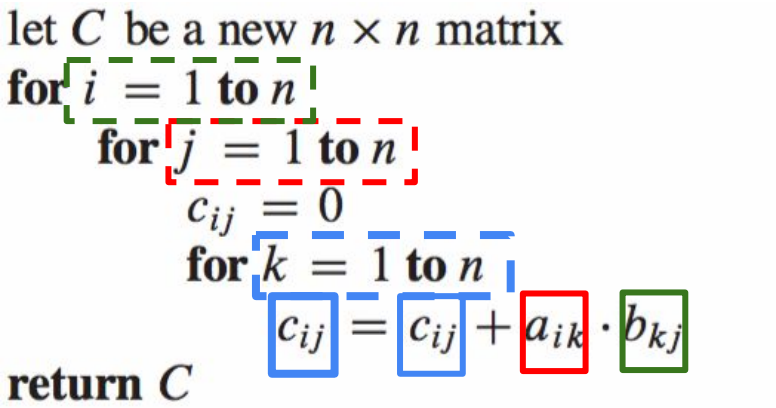
Tile Size Selection

- Consider the temporal and spatial reuse along each dimension of a loop nest
- Smaller tile sizes - Underutilization of cache
- Larger tile sizes - Cache misses
  - Memory footprint of a tile cannot exceed L1 cache capacity
- Programs with multiple loop nests to have different tile sizes for each loop

Method Preliminary: Temporal Reuse Counts

Temporal re-use: (Wolf et al.)

- Choose a loop dimension d, fix **all other** dimensions
- # of memory references that stay invariant to d



Dimension	Re-uses
i	n
j	n
k	2*n

Method Preliminary: Group temporal re-use counts

Group temporal re-use: (Wolf et al.)

- Choose a loop dimension d, fix **all other** dimensions
- # of groups of memory references that exhibit re-use as d iterates forward
- E.g. A[d] = A[d-1] + A[d-2] has factor 3 group temporal re-use in one loop iteration

Method: Setting up the variables

Assume permutable band (loop whose order of dimensions can change freely)

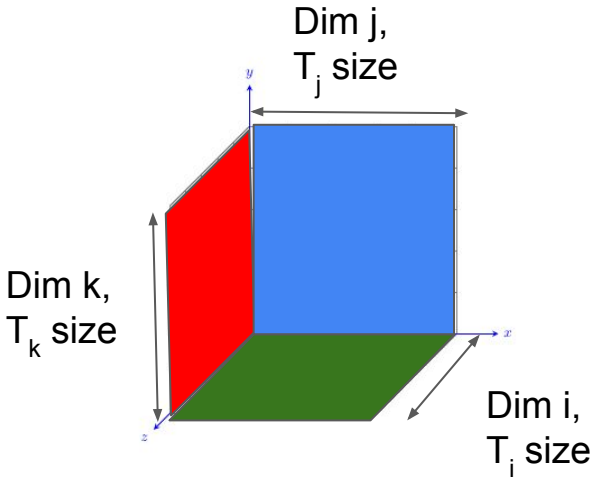
1. Calculate total temporal re-uses per dimension d, normalize to γ_d
   - i: n, j: n, k: 2*n $\rightarrow$ $\gamma_i = 0.5$, $\gamma_j = 0.5$, $\gamma_k = 1$

2. **Assume** target tile sizes for dimension d as: $\mathbf{T_d = \gamma_d * T}$ (**T** is a constant, **to be calculated)**
   - ***Tile sizes are proportionate to re-use*** (greedy performance gains)

Goal: Solve for T

Method: Computing Memory Required

- For each **distinct** memory expression in loop, compute memory “swept” in one tile
  - In our example, we have A[i][k], B[k][j], C[i][j] as expressions
- Sum of these = total memory needed by tile



Expression	Swept memory
A[i][k]	$T_i * T_k$
B[k][j]	$T_k * T_j$
C[i][j]	$T_i * T_j$

Method: The Polynomial and its Solution

- **Total memory needed by tile:** $T_i * T_j + T_j * T_k + T_i * T_k$

  $= (\gamma_i\gamma_j + \gamma_j\gamma_k + \gamma_i\gamma_k) * T^2 =$ ***polynomial_function***(T)

- This is <u>constrained</u> by the size of the cache (in number of elements) = **E**

- ***polynomial_function***$(T) = E$
  - Solve above, use **T** to find $T_i..T_k$

Optimization 1: Exploit Auto-Vectorization

- Compilers like gcc and llvm perform auto-vectorization

- **When does auto-vectorization occur?**
  - Innermost dimension is parallel
  - Innermost dimension has stride-0 or stride-1 access

- Design a heuristic to identify the most vectorization-friendly innermost dimension using **spatial-reuse, temporal-reuse and vectorizability**

Exploit Auto-Vectorization

The heuristic: 2*s + 4*t + 8*v - 16*(a-s-t)

- Promotes **stride-1 accesses** (spatial reuse), **stride-0 accesses** (temporal reuse) and **vectorizability,** and punishes **scatter-gather memory accesses (a-s-t)**

- After finding dimension with highest score,permute it to be the innermost loop

Exploit Auto-Vectorization: MatMul

- Vectorizability is false for k and i:
  - Dimension k carries a dependence as it is an accumulator
  - Dimension i being permuted results in non-contiguous memory accesses
- Dimension j is parallel and has stride-0 and stride-1 accesses only

let C be a new $n \times n$ matrix
for $i = 1$ **to** n
  for $j = 1$ **to** n
    $c_{ij} = 0$
    for $k = 1$ **to** n
      $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
return C

dim	s	t	v	a	score
i	0	1	false	4	-44
j	3	1	true	4	18
k	1	2	false	4	-6

Why dimensions i and k are not vectorizable

$$c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$$

Calculation of s,t,a - Group Spatial and Temporal Reuse

- When a group of references use the same **cache line - Spatial Reuse**
- **Cache line size -** Smallest unit of data transferred between main memory -> CPU cache. 64 bytes for Intel Xeon.
- For 32 bit integers, A[0] - A[15] share the same cache line.
- When a group of references refer to the same location across iterations - **Temporal Reuse**

$$c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$$

Exploit Auto-Vectorization: MatMul

- We want higher tile size for the vector dimension
- Set $T_j = 256$
- Plugging this into the polynomial, the other tile sizes change to **$T_i = 5$, $T_j = 256$, $T_k = 10$**
- Advantage:
  - Permuting dimension j to the innermost loop triggers auto-vectorization
  - Setting a high value for T_j ensures enough iterations can happen in parallel

Optimization 2: Tiling and Parallelism

- High tile sizes for outer parallel dimensions with few iterations => Under utilization
- Assume loop bounds are known
- Calculate total computation size
- Change tile volume T = total_computation_size / #cores
- Impact: Ensures smaller tile sizes and avoids load imbalance

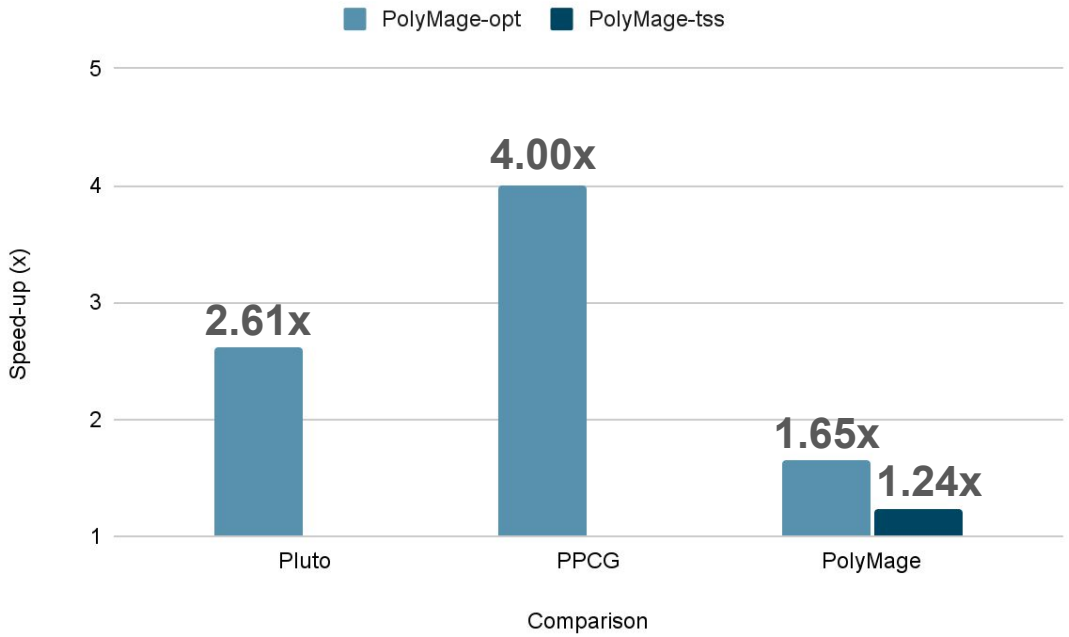
Evaluation

Experimental Setup

- *Baselines*: Pluto, PPCG, PolyMage compilers
- *Implement tile size selection:* Pluto and PolyMage
- *Benchmarks*: compare against PolyBench, DSP, Image Processing.
- *CPU*: Intel Xeon Processor, 16 cores

Evaluation

- **Lin-Alg PolyBench: Speed-ups across the board!**



Evaluation

- **DSP Benchmarks: Mean speedup of 11.8x on SciPy, 2.2x on MATLAB**

Table 7: Execution time (s) and tile size selection time (ms) for digital signal processing benchmarks.

Benchmark	Scipy		Matlab		PolyMage-fft		PolyMage-opt		PolyMage-opt speedup			Tile size selection
	1	16	1	16	1	16	1	16	SciPy	Matlab	PolyMage-fft	time (ms)
Vuvuzela	5.848	8.972	7.633	0.958	13.92	1.123	10.055	0.886	10.13	1.08	1.27	34.51
Unwanted Spectral	0.225	0.294	0.289	0.097	0.234	0.022	0.206	0.021	13.86	4.57	1.04	15.92
geomean									**11.85**	**2.22**	**1.15**	

Evaluation

Other Key Results

- PolyBench whole suite: mean speedup of 1.04x over Pluto

- Image Processing: performs within ±4% of baseline, so no performance regression

Our Reflections on Paper

Strengths	Limitations
Fast and lightweight approach	Less effective when loop bounds unknown at compile time
Accounts for vectorizability	Lesser impact in Pluto since other optimizations (like unroll-and-jam) already provide similar data reuse benefits at the register level
Effective across multiple domains (DSP, LA)	Vectorization scoring is still a somewhat arbitrary heuristic
Significant speedups	More effective dimensionality reuse using better schemes than greedy approach?

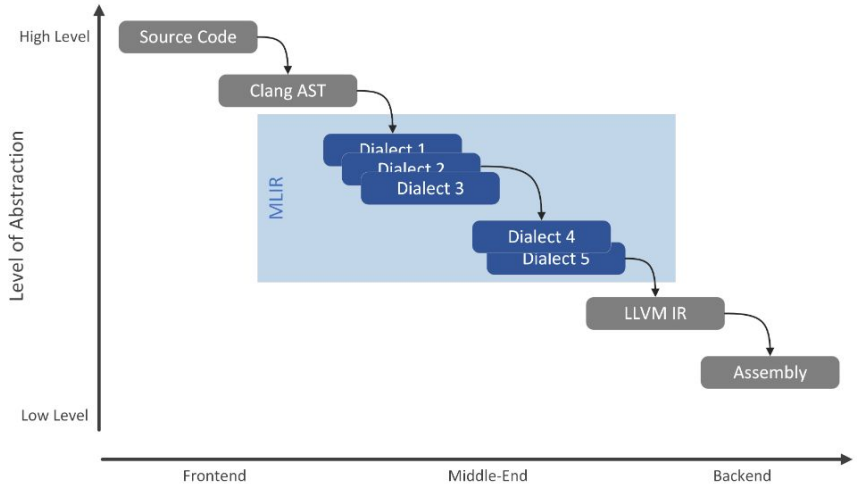
Future Work

MLIR Integration

The authors propose MLIR as the ideal future home for this model, as its extensible infrastructure is perfectly suited to host this optimization.

Runtime Calculation

This integration solves the main limitation. The tile size could be calculated at runtime, just before execution, when the exact problem sizes are finally known.



Conclusion

A Working Model: The paper successfully delivers a fast, generic, and robust tile size selection model that works in practice.

Significant Speedups: It demonstrated clear performance gains

Ready for Adoption: The combination of high performance and negligible compile-time overhead makes it a strong candidate for real-world compilers like GCC, LLVM, and MLIR.

References

Kumudha Narasimhan, Aravind Acharya, Abhinav Baid, and Uday Bondhugula. 2021. A practical tile size selection model for affine loop nests. In Proceedings of the 35th ACM International Conference on Supercomputing (ICS '21). Association for Computing Machinery, New York, NY, USA, 27–39. https://doi.org/10.1145/3447818.3462213

M. Wolf and Monica S. Lam. 1991. A data locality optimizing algorithm. In *ACM SIGPLAN symposium on Programming Languages Design and Implementation*. 30–44.