A Practical Tile Size Selection Model for Affine Loop Nests

Kumudha Narasmimhan [1] Aravind Acharya [2]
Abhinav Baid Uday Bondhugula

Computer Science and Automation
Indian Institute of Science
Bengaluru, India

[1]Codeplay Software Ltd, Edinburgh [2]NVIDIA, Redmond

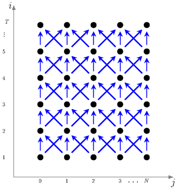
International Conference on Supercomputing 2021

Overview

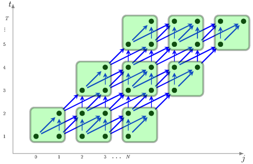
Loop tiling

```
for (i=1; i<T; i++)
  for (j=1; j<N; j++)
    A[i][j] = 0.125 * A[i-1][j-1] +
              0.500 * A[i-1][j] +
              0.125 * A[i-1][j+1];
```

Heat-1d kernel



Iteration space.



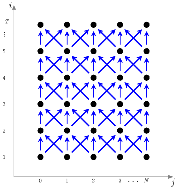
$S(i,j) \rightarrow (i/2, (i+j)/2, i, i+j)$.

Loop tiling improves performance by exploiting reuse in a loop nest.

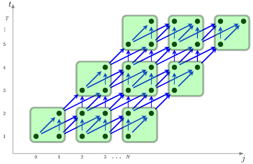
Loop tiling

```
for (i=1; i<T; i++)
  for (j=1; j<N; j++)
    A[i][j] = 0.125 * A[i-1][j-1] +
              0.500 * A[i-1][j] +
              0.125 * A[i-1][j+1];
```

Heat-1d kernel

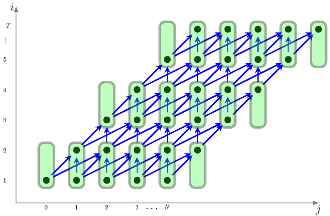


Iteration space.

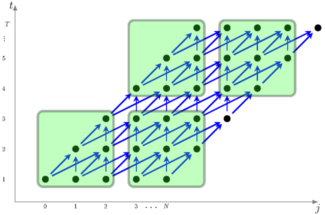


$S(i, j) \rightarrow (i/2, (i+j)/2, i, i+j)$.

Loop tiling improves performance by exploiting reuse in a loop nest.

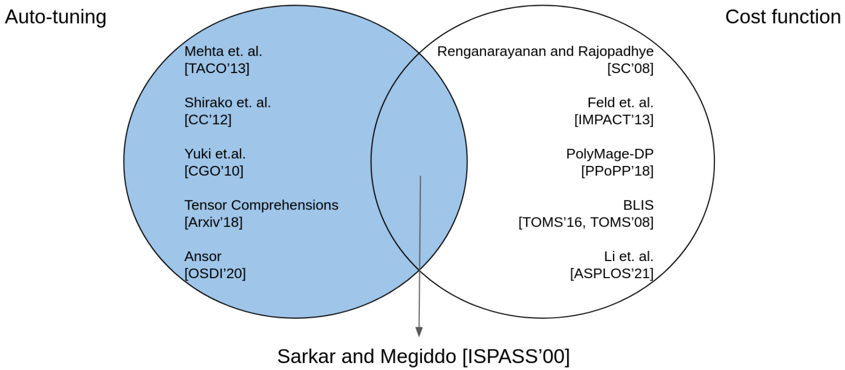


Small tile size $\implies$ Under-utilization.



Large tile size $\implies$ Lesser parallelism.

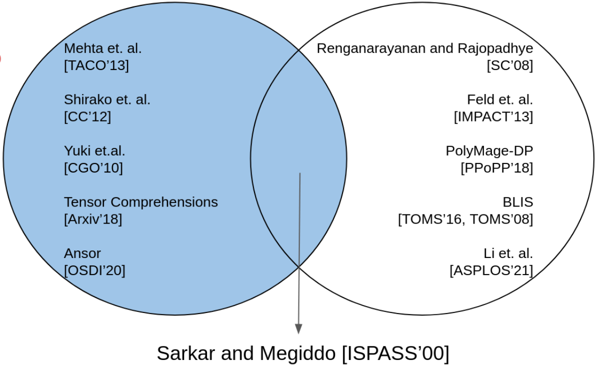
Motivation and Objective



Motivation and Objective

Auto-tuning

Takes long to generate good tile sizes



Cost function

Limited to certain domains

Models complex nonlinear functions

High compile time

Optimizing compilers Pluto and PPCG use default tile sizes and Performance improvement of **6.4**× over default tile sizes in certain cases.

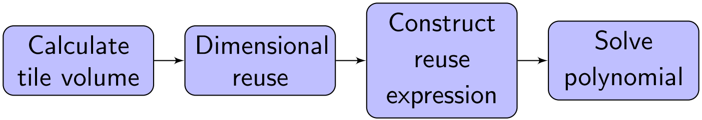
Outline

Tile Size Selection Model

Objectives of the proposed tile size selection model

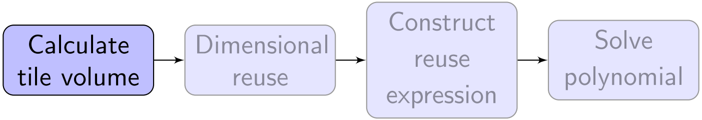
- Work for arbitrary **affine access**.
- Compute tile sizes **quickly**.
- Tile any level in the memory hierarchy.
- Consider effects of tiling on **parallelism**
- Note our aim is not to find the optimal tile size but **good tile sizes** that can be obtained **quickly**.

Tile Size Selection Model: Intuition



- Tiling is profitable when there is simultaneous reuse of data along multiple dimensions
- Larger tile sizes along dimensions with more reuse, utilizes the cache effectively.
- Hence, in our model, tile sizes are proportional to reuse along the dimension

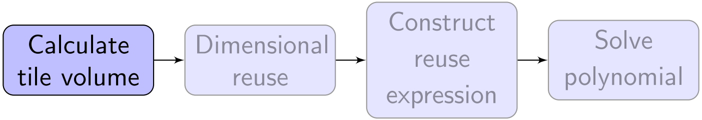
Tile Size Selection Model: Inputs



Input to the tile size selection model:

- cache size (L1 or L2)
- datatype of the element stored
  - $NumElementsInCache(C) = cacheSize / elementSize$

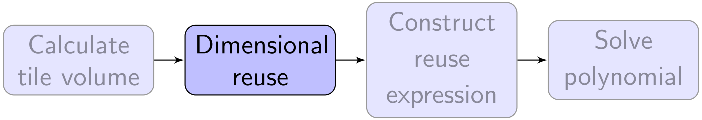
Tile Size Selection Model: Inputs



Input to the tile size selection model:

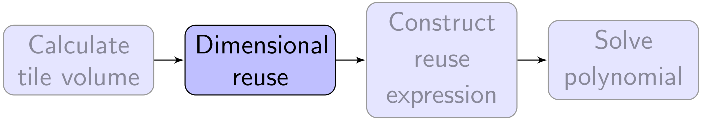
- cache size (L1 or L2)
- datatype of the element stored
  - $NumElementsInCache(C) = cacheSize/elementSize$
- problem size
- number of cores
  - Used to calculate effective computation, to avoid load imbalance

Tile Size Selection Model: Dimensional Reuse



Dimensional Reuse along i (γ_i)
$=$ Number of access which have temporal reuse along i

Tile Size Selection Model: Dimensional Reuse

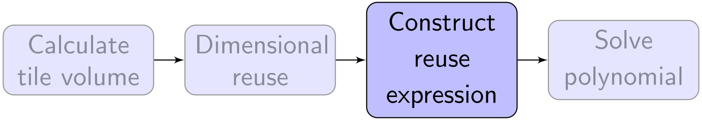


Dimensional Reuse along i (γ_i)
= Number of access which have temporal reuse along i

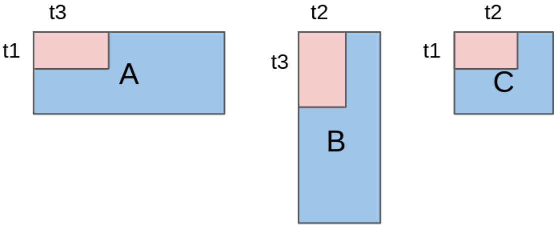
```
for(int i = 0; i < N; i++)
  for(int j = 0; j < N; j++)
    for(int k = 0; k < N; k++)
      C[i][j] += A[i][k] * B[k][j];
```

- Reuse along $\gamma_i = 1$, $\gamma_j = 1$ and $\gamma_k = 2$
- Tile size for k will be twice that of i or j.

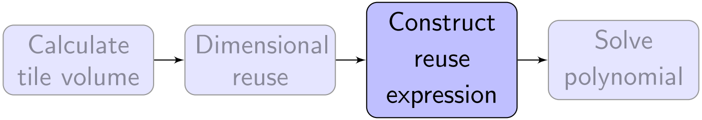
Tile Size Selection Model: Reuse expression



The tile volume should fit in the cache.

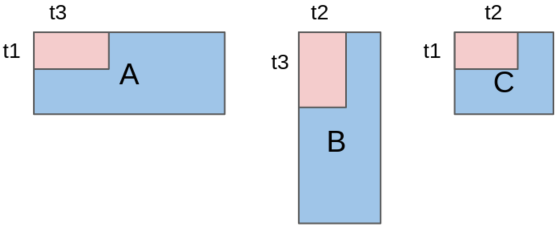


Tile Size Selection Model: Reuse expression

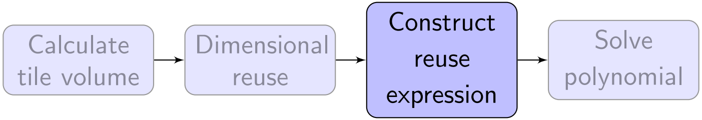


The tile volume should fit in the cache.

$$t_1 * t_2 + t_2 * t_3 + t_3 * t_1 = C.$$

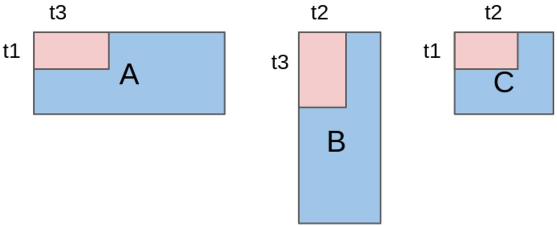


Tile Size Selection Model: Reuse expression



The tile volume should fit in the cache.

$$t_1 * t_2 + t_2 * t_3 + t_3 * t_1 = C.$$

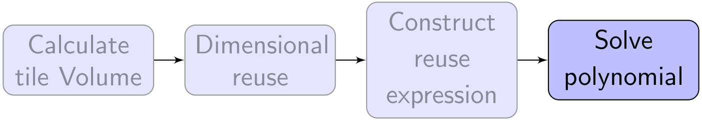


In our model, tile sizes are proportional to dimensional reuse. Therefore,

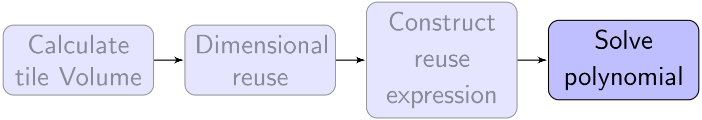
$$t_i = \gamma_i \times \tau, \text{ for every dimension } i.$$

$$(\gamma_i * \gamma_j + \gamma_j * \gamma_k + \gamma_k * \gamma_i) * \tau^2 = C.$$

Tile Size Selection Model: Reuse expression



Tile Size Selection Model: Reuse expression



$$(\gamma_i * \gamma_j + \gamma_j * \gamma_k + \gamma_k * \gamma_i) * \tau^2 = C.$$

$$(0.5 * 0.5 + 0.5 * 1.0 + 1.0 * 0.5) * \tau^2 = 4096.$$
$$\implies 1.25 * \tau^2 - 4096 = 0.$$

Solving for τ, tile sizes can be computed as, $t_1 = 28$, $t_2 = 28$ and $t_3 = 57$.

Tile Size Selection Model: Reuse Expression

Reuse Expressions for arbitrary affine accesses

memory access	no. of distinct accesses w.r.t tile size	no. of distinct accesses w.r.t dimensional reuse
$a[i]$	τ_i	$\gamma_i * \tau$
$a[\alpha * i]$	τ_i	$\gamma_i * \tau$
$a[i + j]$	$\tau_i + \tau_j$	$(\gamma_i + \gamma_j) * \tau$
$a[i - j]$	$\tau_i + \tau_j$	$(\gamma_i + \gamma_j) * \tau$
$a[i][j]$	$\tau_i * \tau_j$	$(\gamma_i * \gamma_j) * \tau^2$

Outline

Intra-tile optimization: Matmul Example

```
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];
```

- loop k carries a dependence $\implies$ Not parallel $\implies$ Not vectorizable
- loop i has non contiguous accesses for arrays C and A $\implies$ Not vectorizable

Intra-tile optimization: Matmul Example

```
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];
```

- loop k carries a dependence $\implies$ Not parallel $\implies$ Not vectorizable
- loop i has non contiguous accesses for arrays C and A $\implies$ Not vectorizable
- $score = score + (2 * s) + (4 * t) + (8 * v) - (16 * (a - s - t))$

Intra-tile optimization: Matmul Example

```
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];
```

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

- loop k carries a dependence $\implies$ Not parallel $\implies$ Not vectorizable
- loop i has non contiguous accesses for arrays C and A $\implies$ Not vectorizable
- $score = score + (2 * s) + (4 * t) + (8 * v) - (16 * (a - s - t))$

Intra-tile optimization: Matmul Example

```
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];
```

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

- loop k carries a dependence $\Longrightarrow$ Not parallel $\Longrightarrow$ Not vectorizable
- loop i has non contiguous accesses for arrays C and A $\Longrightarrow$ Not vectorizable
- $score = score + (2 * s) + (4 * t) + (8 * v) - (16 * (a - s - t))$
- loop j has the highest score and is selected as the inner-most dimension

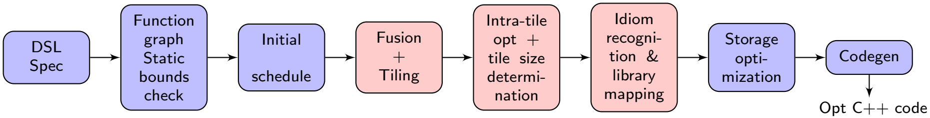
Tile Size Selection Model

- Assigns a constant large tile size for the vectorizable dimension

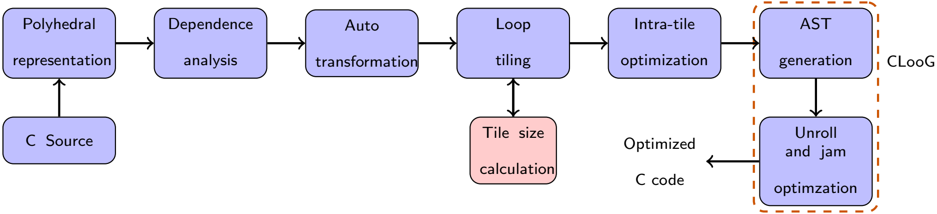
$$(\gamma_i + \gamma_k) * 256 * \tau + (\gamma_k * \gamma_i) * \tau^2 = C.$$

$$(0.5 + 1.0)256 * \tau + (1.0 * 0.5) * \tau^2 = (4096).$$
$$\implies 0.5 * \tau^2 + 384 * \tau - 4096 = 0.$$

Implementation



Modified compiler flow in PolyMage.



Modified compiler flow in Pluto.

Outline

Experimental Setup

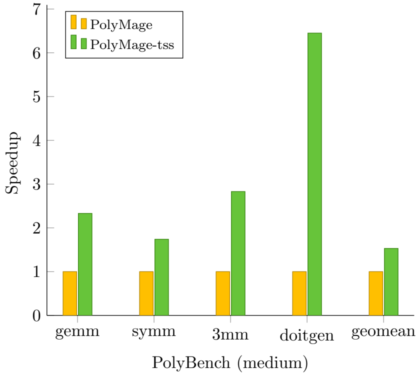
Benchmarks

- 26 benchmarks from PolyBench.
- 2 Digital Signal Processing filters.
- Image Processing benchmarks.

Experimental setup

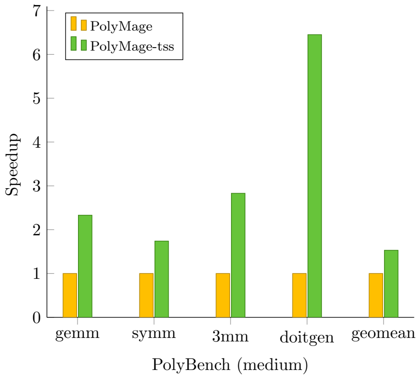
Processors	Intel(R) Xeon(R) Silver 4110 CPU @ 2.10 GHz
Cores	16 (8 per socket)
Private caches	32 KB L1 cache, 1 MB L2 cache
Memory	256 GB DDR4
Matlab version	9.9.0.1524771 (R2020b)
Scipy version	1.0.0
Compiler	Intel C/C++ (icc/icpc) 19.1.2.254
Compiler flags	-O3 -xhost -qopenmp -fma -ipo

PolyBench Results

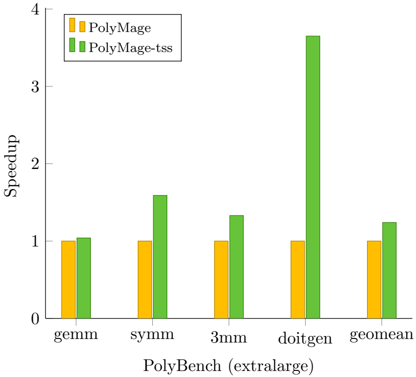


Geomean speedup of 1.53× over PolyMage

PolyBench Results

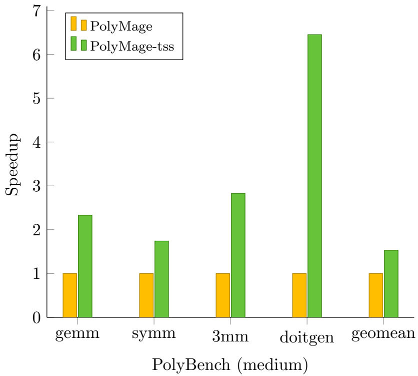


Geomean speedup of 1.53× over PolyMage

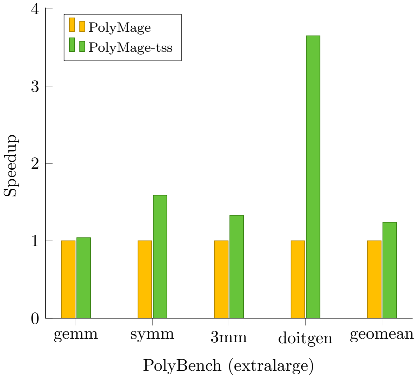


Geomean speedup of 1.24× over PolyMage

PolyBench Results

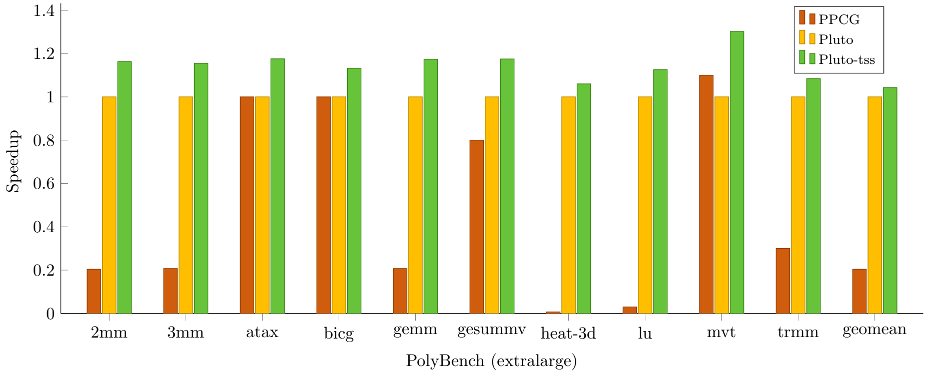


Geomean speedup of 1.53× over PolyMage
Max tile size selection time - 13*ms*.



Geomean speedup of 1.24× over PolyMage

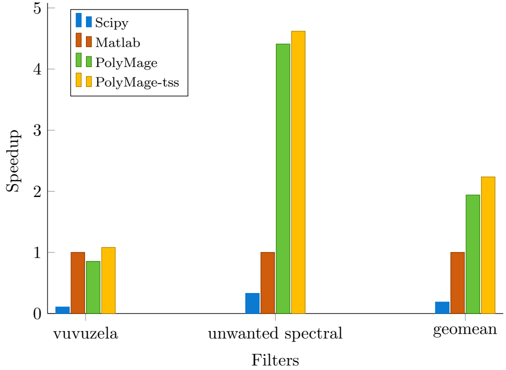
PolyBench Results



Geomean speedup of 1.04$\times$ over Pluto.

Max tile size selection time - 2ms.

DSP Benchmark Results



- Upsampling operations are mapped to FFTs.
- Geomean speedup of 11.8× over SciPy and 2.2× over Matlab.
- Max tile size selection time - 35*ms*.

Conclusion

- PolyBench
  - Mean speedup of 1.04× (max 1.3×) over Pluto for the entire PolyBench.
  - Mean speedup of 1.24× (max 3.65×) for linear algebra benchmarks.
- Digital Signal Processing
  - Geomean speedup of 11.8× over Intels Scipy and 2.2× over MATLAB
- Image Processing benchmarks
  - We get similar tile sizes as Jangda et. al. [PPoPP'18]
- Code available - https://github.com/bondhugula/pluto

Conclusion

- Proposed a **simple, fast, and practical approach** for tile size selection
- **Model-driven** and gives good performance improvement - frees existing tools from using hardcoded sizes.
  - Geomean speedup of **1.24**$\times$ over PolyMage, **1.04**$\times$ over Pluto, and **5.11**$\times$ over PPCG.
  - Geomean speedup of **11.8**$\times$ over Intels Scipy and **2.2**$\times$ over MATLAB for DSP filters
- **Neglible compile time overhead** of model makes it suitable for incorporation in a general-purpose compiler infrastructure like MLIR.
- The model can be easily extended to **multi-level tiling**

- Proposed a **simple, fast, and practical approach** for tile size selection
- **Model-driven** and gives good performance improvement - frees existing tools from using hardcoded sizes.
  - Geomean speedup of **1.24**× over PolyMage, **1.04**× over Pluto, and **5.11**× over PPCG.
  - Geomean speedup of **11.8**× over Intels Scipy and **2.2**× over MATLAB for DSP filters
- **Neglible compile time overhead** of model makes it suitable for incorporation in a general-purpose compiler infrastructure like MLIR.
- The model can be easily extended to **multi-level tiling**

We would like to thank the reviewers for their feedback that helped improve the paper significantly.