To Pack or Not to Pack: A Generalized Packing Analysis and Transformation

Caio S. Rohwedder - Nathan Henderson
João P. L. de Carvalho - Yufei Chen - J. Nelson Amaral

Motivation

High-Performance GEMM: Tiling and Packing

$$\begin{bmatrix}
a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 \\
b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & b_8 & b_9 \\
c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 \\
d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 & d_9 \\
e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 & e_9 \\
f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 & f_8 & f_9 \\
g_1 & g_2 & g_3 & g_4 & g_5 & g_6 & g_7 & g_8 & g_9 \\
h_1 & h_2 & h_3 & h_4 & h_5 & h_6 & h_7 & h_8 & h_9 \\
i_1 & i_2 & i_3 & i_4 & i_5 & i_6 & i_7 & i_8 & i_9 \\
j_1 & j_2 & j_3 & j_4 & j_5 & j_6 & j_7 & j_8 & j_9
\end{bmatrix}$$

High-Performance GEMM: Tiling and Packing

$$\begin{bmatrix}
a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 \\
b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & b_8 & b_9 \\
c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 \\
d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 & d_9 \\
e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 & e_9 \\
f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 & f_8 & f_9 \\
g_1 & g_2 & g_3 & g_4 & g_5 & g_6 & g_7 & g_8 & g_9 \\
h_1 & h_2 & h_3 & h_4 & h_5 & h_6 & h_7 & h_8 & h_9 \\
i_1 & i_2 & i_3 & i_4 & i_5 & i_6 & i_7 & i_8 & i_9 \\
j_1 & j_2 & j_3 & j_4 & j_5 & j_6 & j_7 & j_8 & j_9
\end{bmatrix}$$

High-Performance GEMM: Tiling and Packing

$$\begin{bmatrix}
a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 \\
b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & b_8 & b_9 \\
c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 \\
d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 & d_9 \\
e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 & e_9 \\
f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 & f_8 & f_9 \\
g_1 & g_2 & g_3 & g_4 & g_5 & g_6 & g_7 & g_8 & g_9 \\
h_1 & h_2 & h_3 & h_4 & h_5 & h_6 & h_7 & h_8 & h_9 \\
i_1 & i_2 & i_3 & i_4 & i_5 & i_6 & i_7 & i_8 & i_9 \\
j_1 & j_2 & j_3 & j_4 & j_5 & j_6 & j_7 & j_8 & j_9
\end{bmatrix}$$

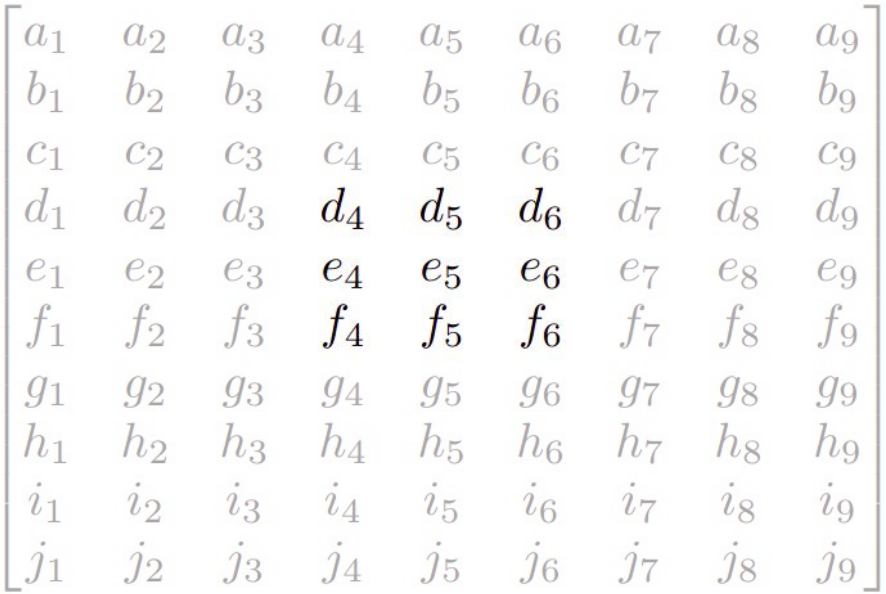
High-Performance GEMM: Tiling and Packing

$$\begin{bmatrix}
a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 \\
b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & b_8 & b_9 \\
c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 \\
d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 & d_9 \\
e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 & e_9 \\
f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 & f_8 & f_9 \\
g_1 & g_2 & g_3 & g_4 & g_5 & g_6 & g_7 & g_8 & g_9 \\
h_1 & h_2 & h_3 & h_4 & h_5 & h_6 & h_7 & h_8 & h_9 \\
i_1 & i_2 & i_3 & i_4 & i_5 & i_6 & i_7 & i_8 & i_9 \\
j_1 & j_2 & j_3 & j_4 & j_5 & j_6 & j_7 & j_8 & j_9
\end{bmatrix}$$

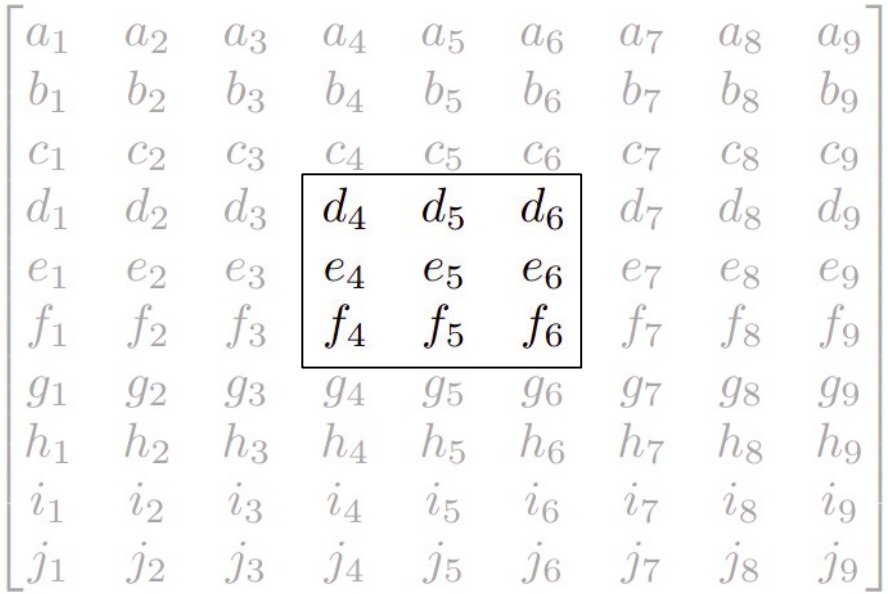
High-Performance GEMM: Tiling and Packing

$$\begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 \\ b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & b_8 & b_9 \\ c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 \\ d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 & d_9 \\ e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 & e_9 \\ f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 & f_8 & f_9 \\ g_1 & g_2 & g_3 & g_4 & g_5 & g_6 & g_7 & g_8 & g_9 \\ h_1 & h_2 & h_3 & h_4 & h_5 & h_6 & h_7 & h_8 & h_9 \\ i_1 & i_2 & i_3 & i_4 & i_5 & i_6 & i_7 & i_8 & i_9 \\ j_1 & j_2 & j_3 & j_4 & j_5 & j_6 & j_7 & j_8 & j_9 \end{bmatrix}$$

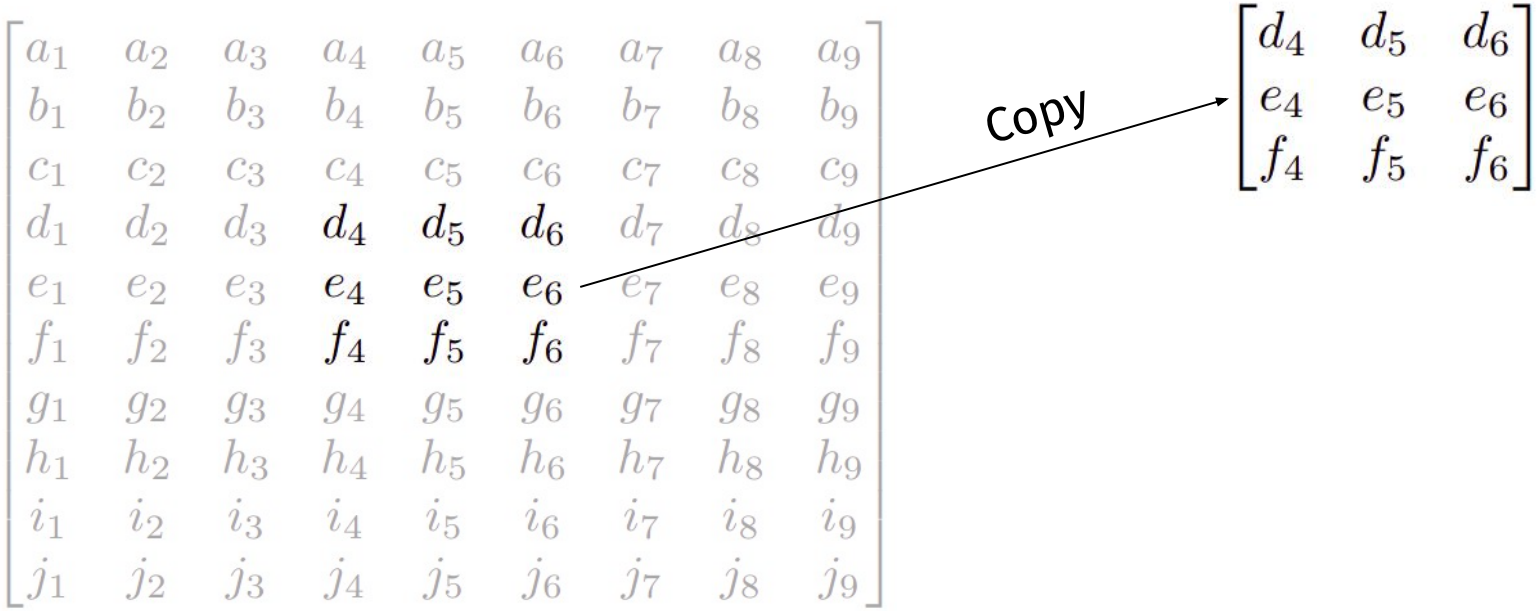
High-Performance GEMM: Tiling and Packing



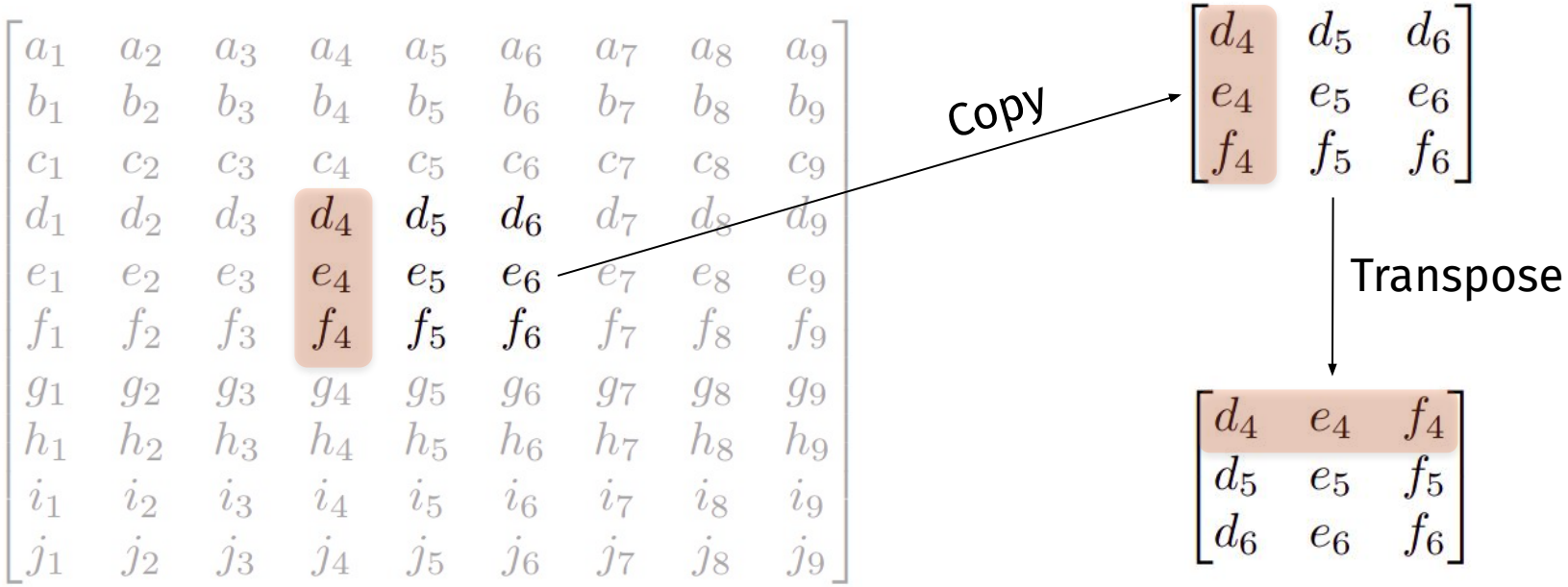
High-Performance GEMM: Tiling and Packing



High-Performance GEMM: Tiling and Packing



High-Performance GEMM: Tiling and Packing



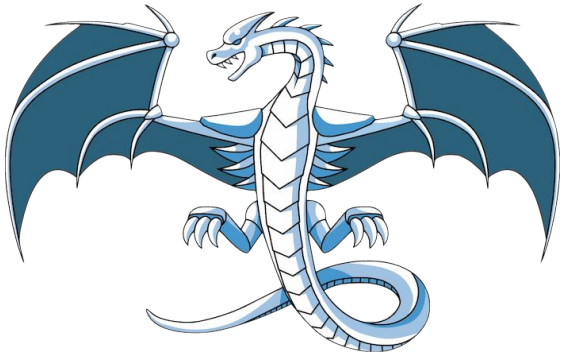
Current state of Loop Optimizers

bondhugula/**pluto**

Pluto: An automatic polyhedral parallelizer and locality optimizer

MLIR

tvm



polly

halide/**Halide**

a language for fast, portable data-parallel computation

Current state of Loop Optimizers

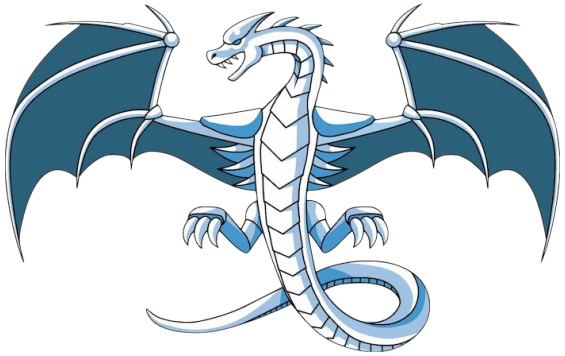
- Tiling
- Unrolling
- Interchanging
- Fusion
- *etc.*

bondhugula/**pluto**

Pluto: An automatic polyhedral parallelizer and locality optimizer

MLIR

tvm



polly

halide/**Halide**

a language for fast, portable data-parallel computation

Packing? No.

Packing? No.

Except for:

- GEMM-specific

Packing? No.

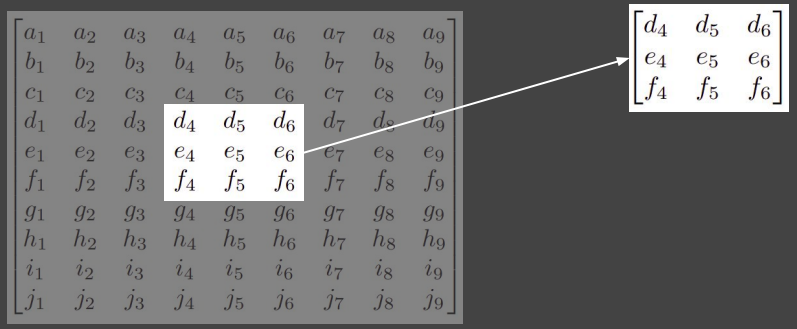
Except for:

- **GEMM-specific**
- **Handcrafted**
- **Autotuned**

We propose:

- Compiler-level packing to generic inputs

- Analysis to determine what and where to pack

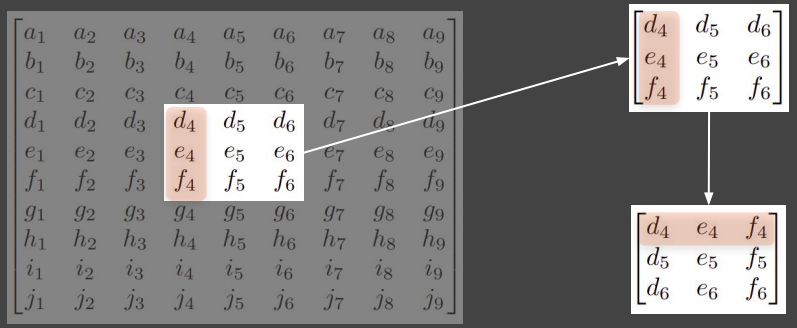


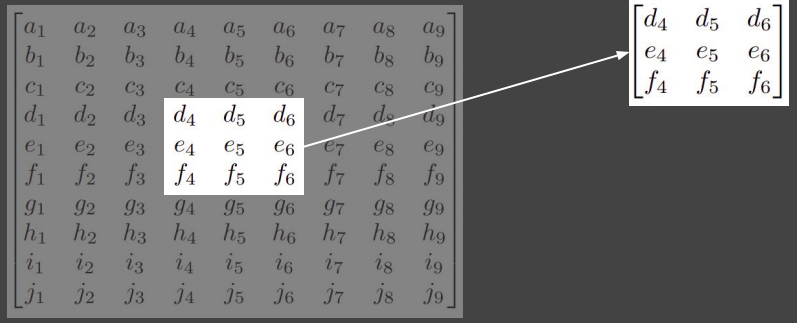
Packing pros:

- Self-interference misses
- TLB entries
- Vectorization

Packing pros:

- Self-interference misses
- TLB entries
- Vectorization





Packing cons:

- Copy overhead
- Naive approach → Slowdown

GPAT: Generalized Packing Analysis and Transformation

A Simple Example

3D Tensor Contraction

```
for(i=0; i<50; i++)
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A[k][l][i]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

3D Tensor Contraction

Pack tensor **A**
targeting loop **ForJ**

```
for(i=0; i<50; i++)
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A[k][l][i]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

3D Tensor Contraction

Pack tensor **A**
targeting loop **ForJ**:

1. Insert packing loop
2. Substitute A for A’
3. Insert unpacking loop

(data-layout change)

```
for(i=0; i<50; i++)
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A[k][l][i]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

3D Tensor Contraction

Pack tensor **A**
targeting loop **ForJ**:

1. Insert packing loop
2. Substitute A for A’
3. Insert unpacking loop

(data-layout change)

```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A[k][l][i]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

3D Tensor Contraction

Pack tensor **A**
targeting loop **ForJ**:

1. Insert packing loop
2. Substitute A for A’
3. Insert unpacking loop

(data-layout change)

```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A[k][l][i]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

3D Tensor Contraction

Pack tensor **A**
targeting loop **ForJ**:

1. Insert packing loop
2. Substitute A for A’
3. Insert unpacking loop

(data-layout change)

```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A’[0][k][l]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

3D Tensor Contraction

Pack tensor **A**
targeting loop **ForJ**:

1. Insert packing loop
2. Substitute A for A’
3. Insert unpacking loop

(data-layout change)

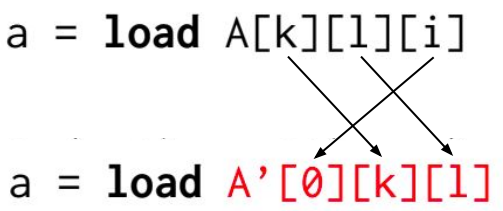
```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A’[0][k][l]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

3D Tensor Contraction

Pack tensor **A**
targeting loop **ForJ**:

1. Insert packing loop
2. Substitute A for A’
3. Insert unpacking loop

(data-layout change)



Packing Analysis

Notation

Loop-tensor pair: **packing candidate**

```
for(i=0; i<50; i++)
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A[k][l][i]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

Notation

Loop-tensor pair: **packing candidate**

```
for(i=0; i<50; i++)
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A[k][l][i]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

Notation

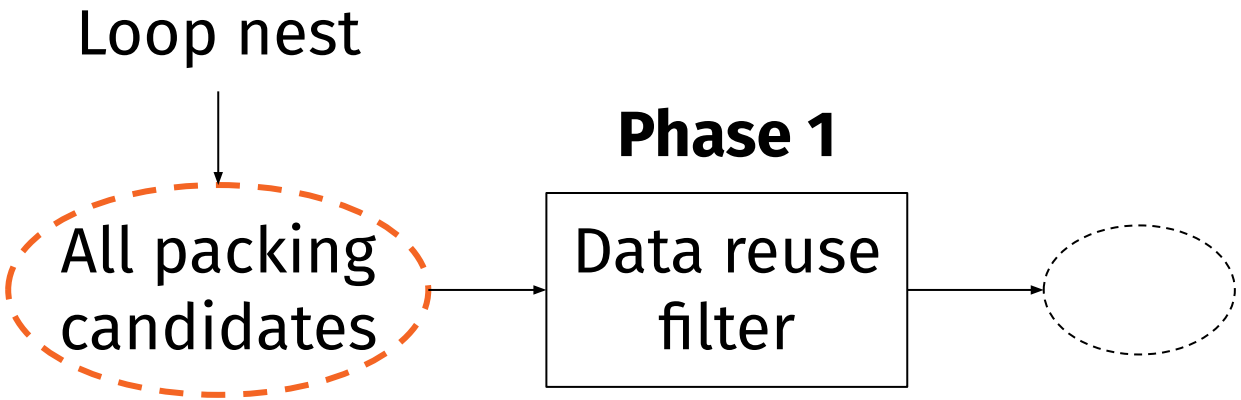
Loop-tensor pair: **packing candidate**

```
for(i=0; i<50; i++)
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A[k][l][i]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

Notation

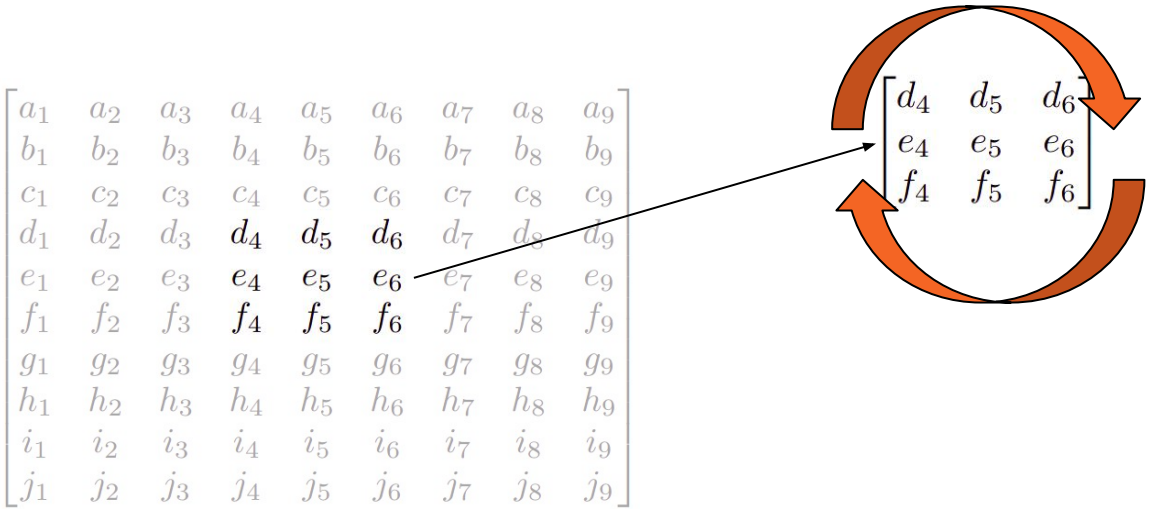
Loop-tensor pair: **packing candidate**

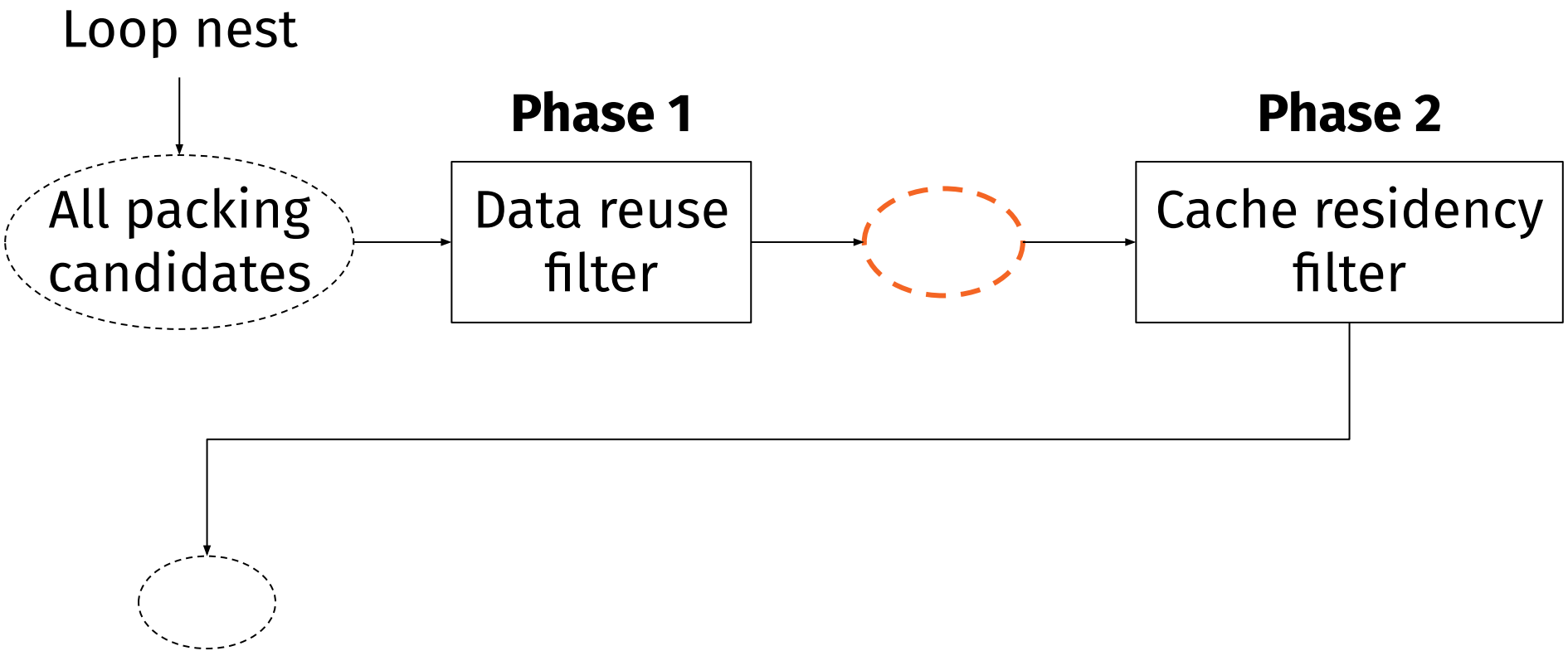
```
for(i=0; i<50; i++)
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A[k][l][i]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```


Loop nest
Phase 1
All packing candidates
Data reuse filter

Phase 1 - Data Reuse Filter

Overcome copying overhead with **reuse**



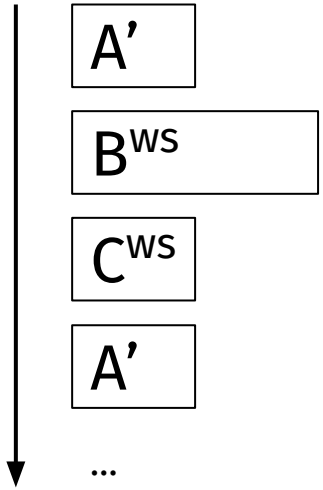

Loop nest
Phase 1
Phase 2
All packing candidates
Data reuse filter
Cache residency filter

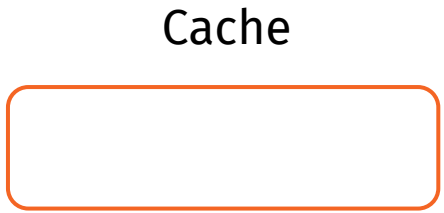
Phase 2 - Cache Residency Filter

Don’t evict T’ from **cache**

Phase 2 - Cache Residency Filter

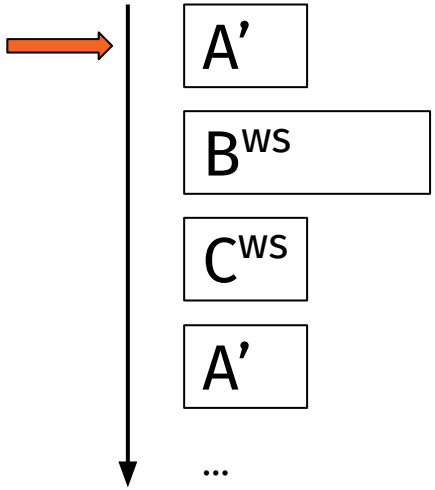
Don’t evict T’ from **cache**

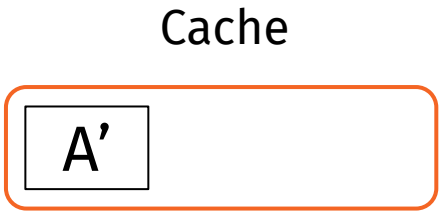




Phase 2 - Cache Residency Filter

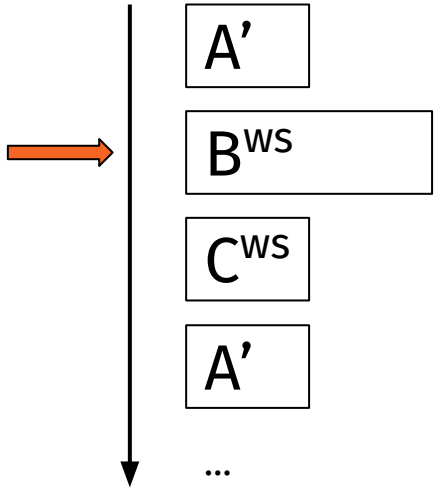
Don’t evict T’ from **cache**

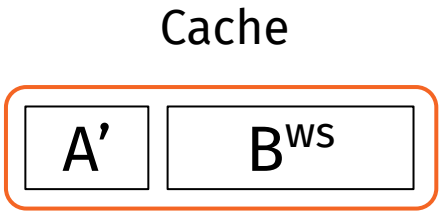




Phase 2 - Cache Residency Filter

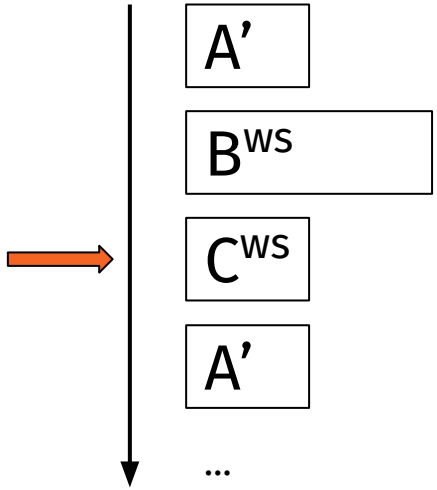
Don’t evict T’ from **cache**



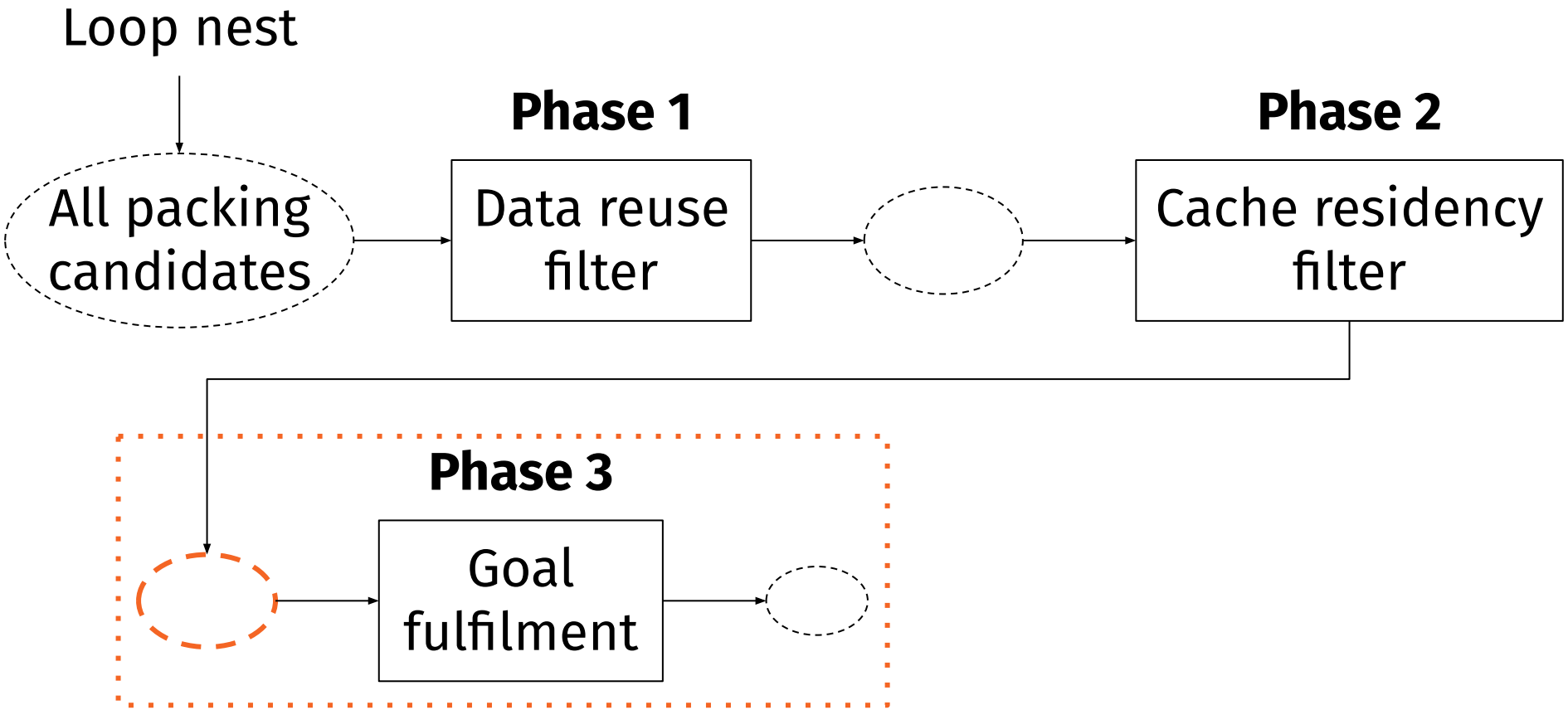


Phase 2 - Cache Residency Filter

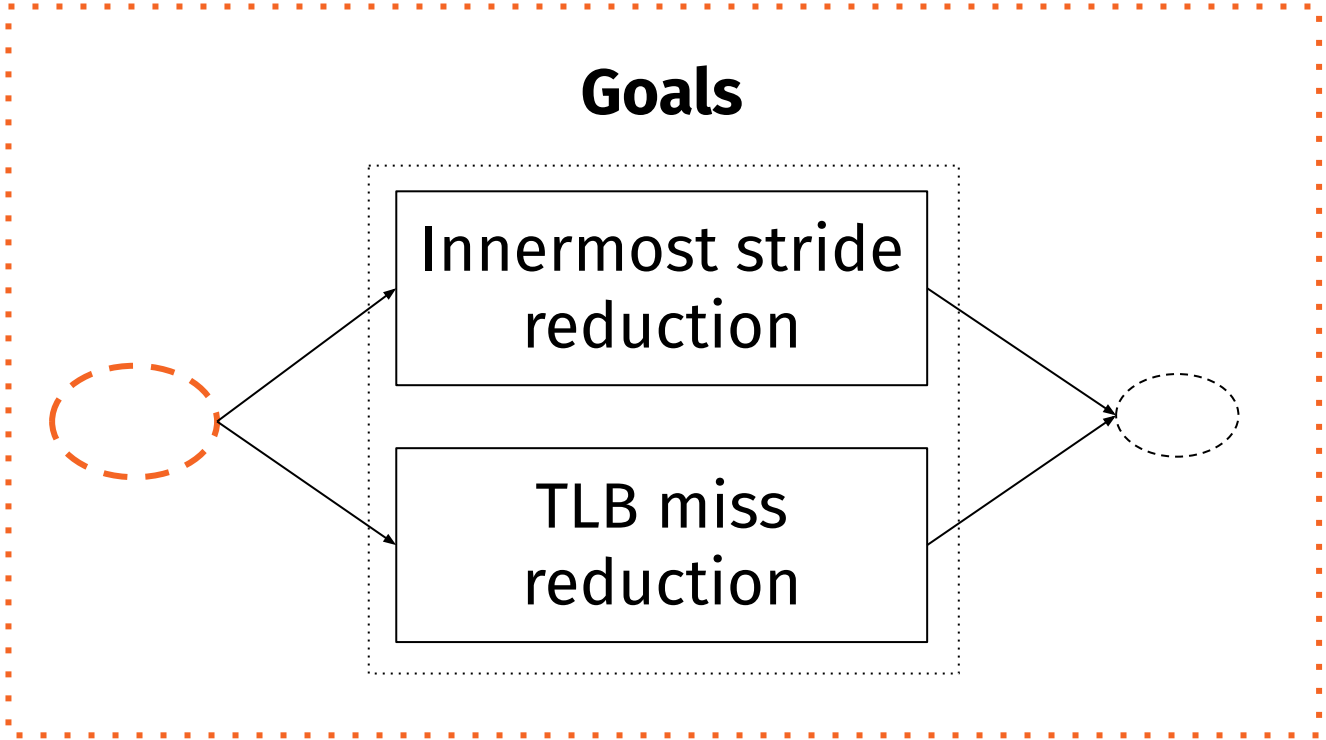
Don’t evict T’ from **cache**



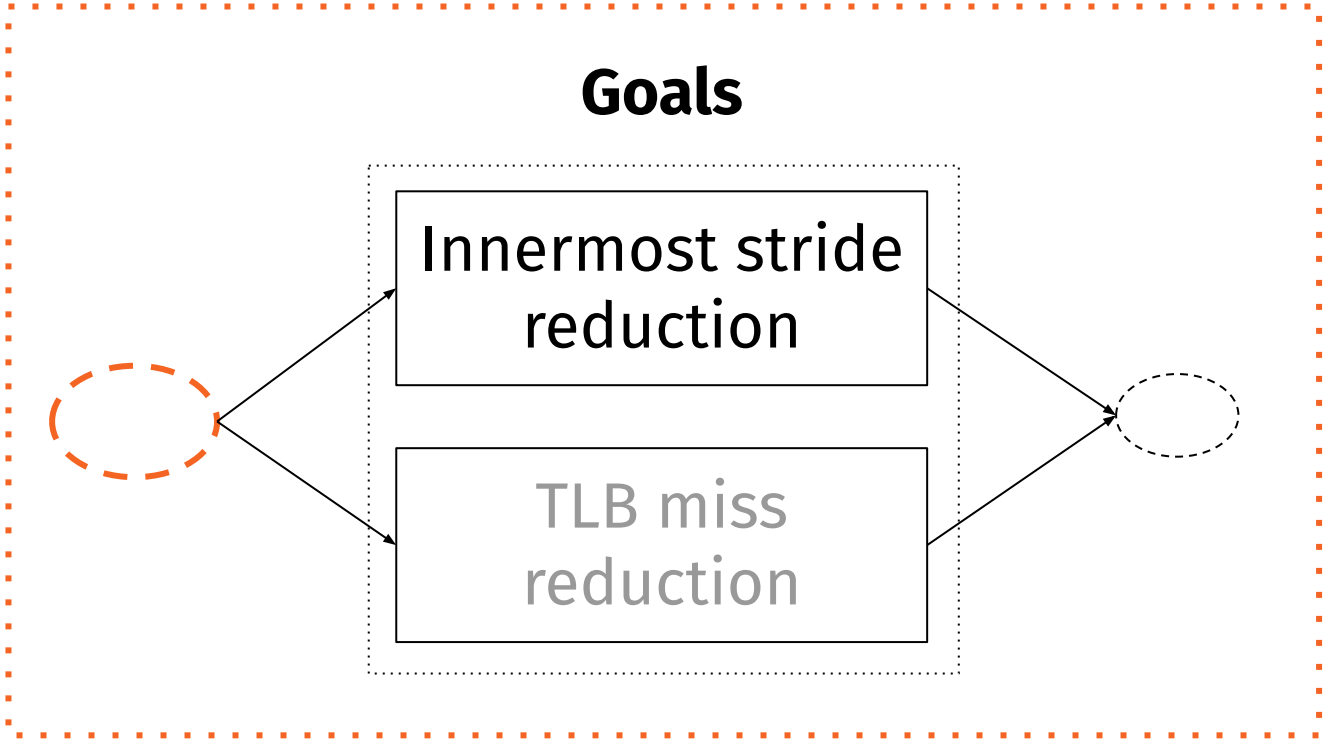
Cache


Loop nest
Phase 1
Phase 2
All packing candidates
Data reuse filter
Cache residency filter
Phase 3
Goal fulfilment

Phase 3 - Goal Fulfilment



Phase 3 - Goal Fulfilment



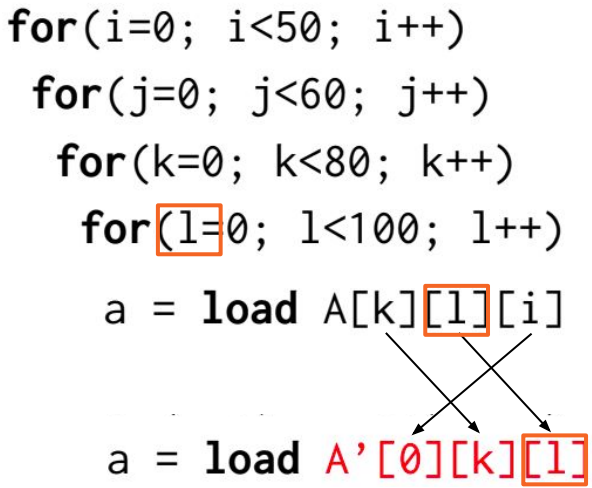
Phase 3 - Innermost Stride Reduction

Data-layout change reduces stride at an **innermost** loop?

```
for(i=0; i<50; i++)
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A[k][l][i]
```

Phase 3 - Innermost Stride Reduction

Data-layout change reduces stride at an **innermost** loop?



Phase 3 - Innermost Stride Reduction

Data-layout change reduces stride at an **innermost** loop?

- Cache locality
- Vectorization

Phase 3 - Innermost Stride Reduction

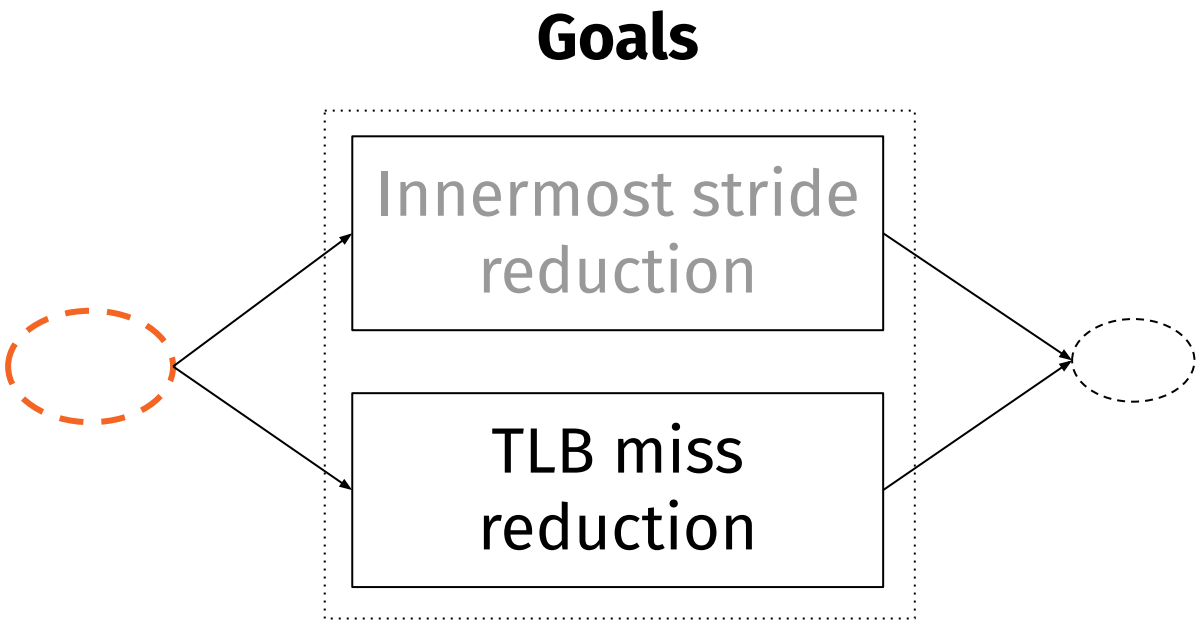
Data-layout change reduces stride at an **innermost** loop?

- Cache locality
- Vectorization

What about stride reduction at other loops?

- Less TLB entries

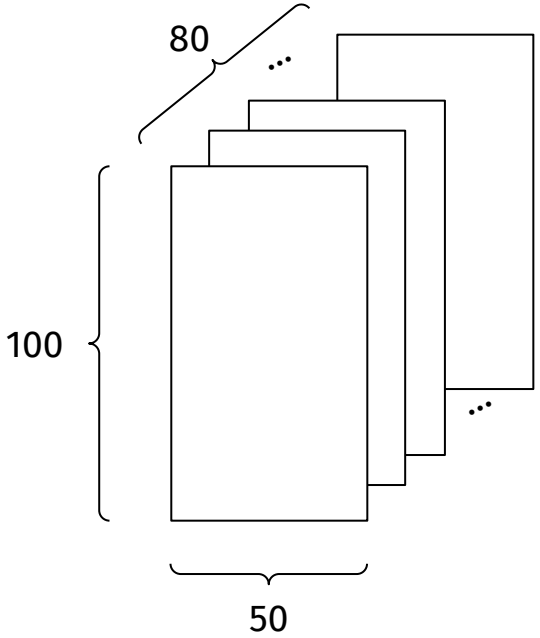
Phase 3 - Goal Fulfilment



Phase 3 - TLB Miss Reduction

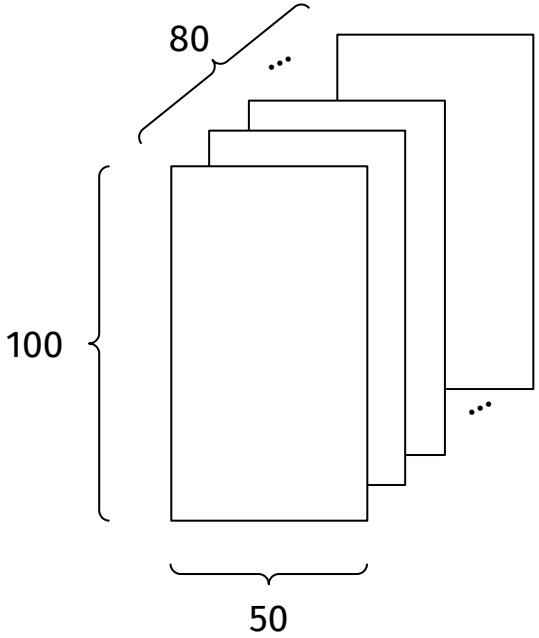
Packing reduces TLB entries
in a loop below L1 dTLB capacity?

Phase 3 - TLB Miss Reduction

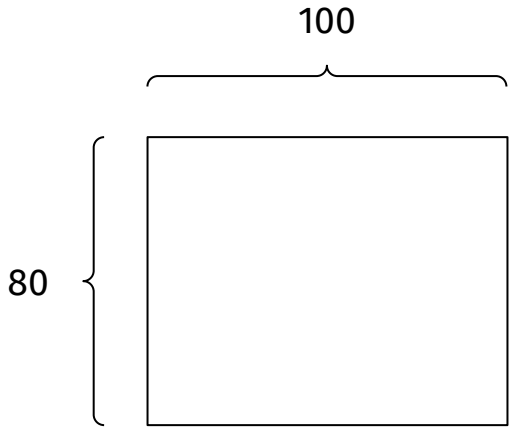


$A_{80x100x50}$

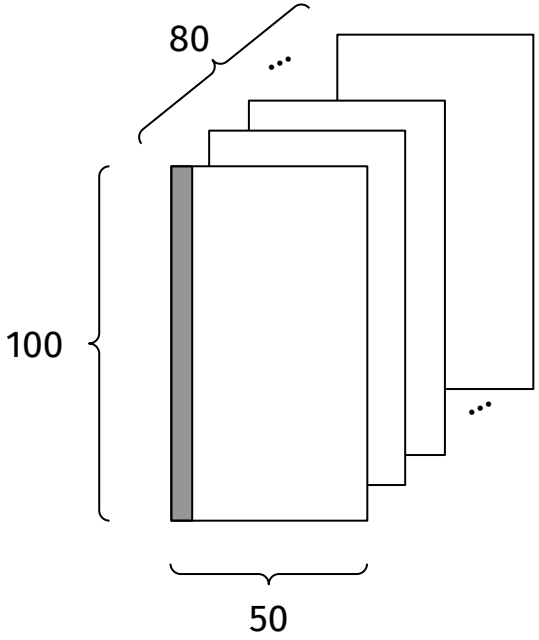
Phase 3 - TLB Miss Reduction



$$A_{80x100x50} \longrightarrow A'_{1x80x100}$$

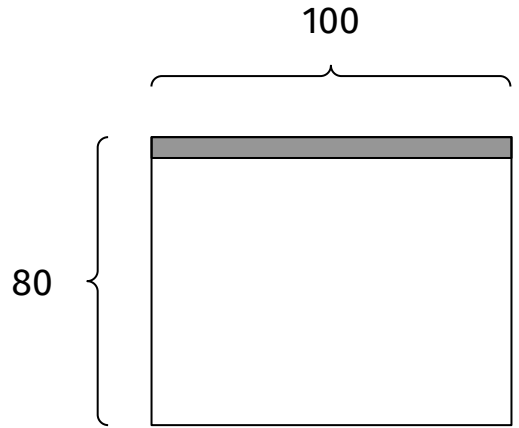


Phase 3 - TLB Miss Reduction



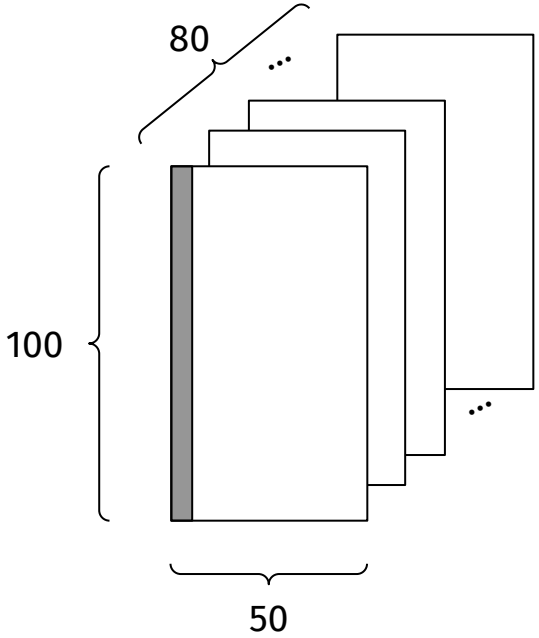
$$A_{80x100x50} \longrightarrow A'_{1x80x100}$$

$$A_{1x100x1} \longrightarrow A'_{1x1x100}$$



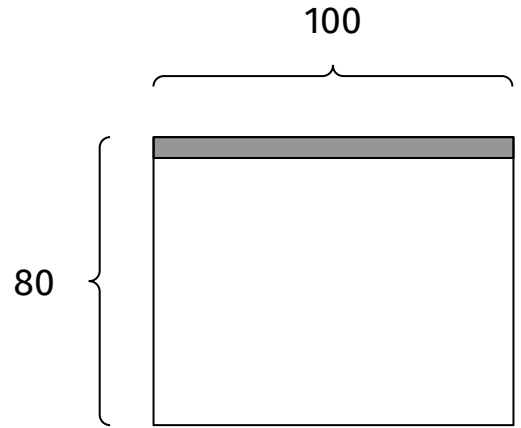
Phase 3 - TLB Miss Reduction

1 TLB entry = 50 elements



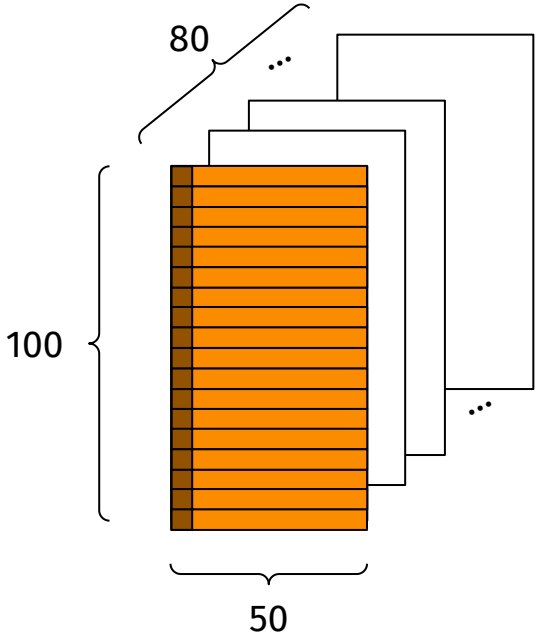
$$A_{80x100x50} \longrightarrow A'_{1x80x100}$$

$$A_{1x100x1} \longrightarrow A'_{1x1x100}$$



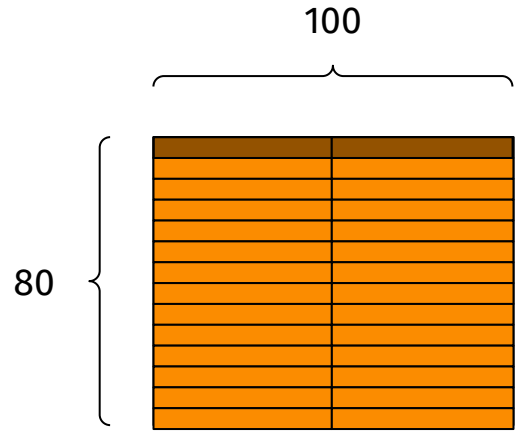
Phase 3 - TLB Miss Reduction

1 TLB entry = 50 elements



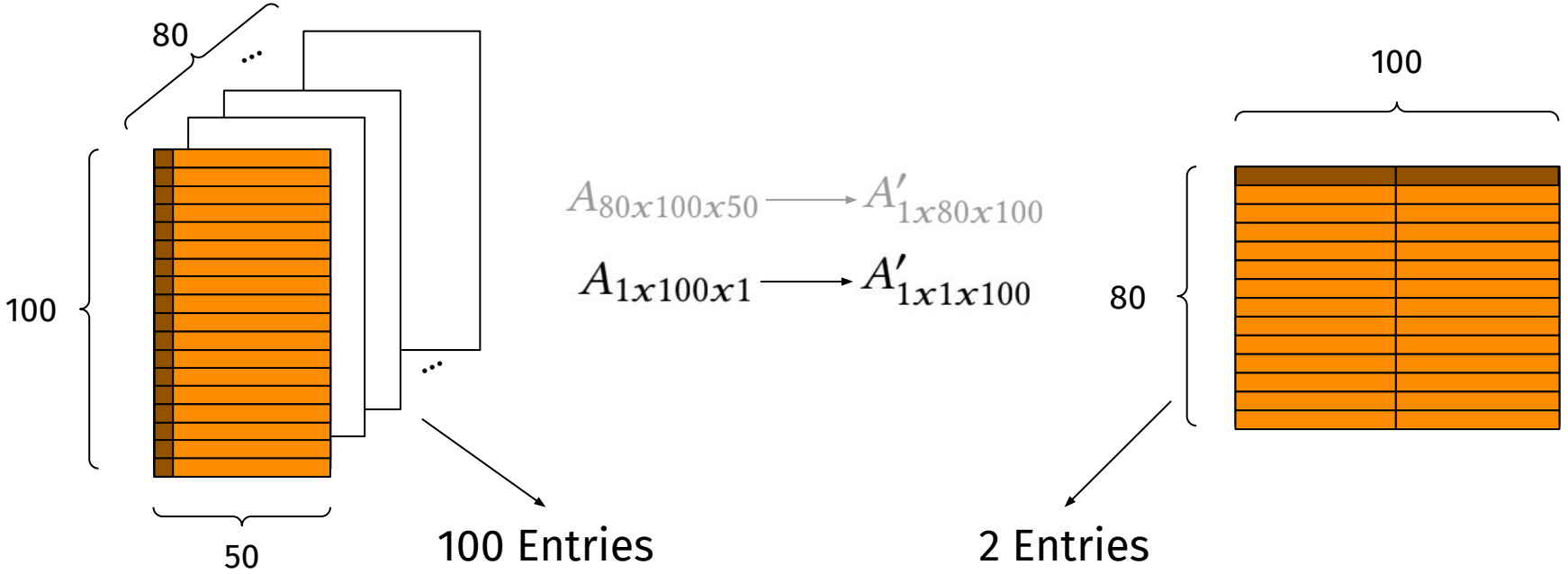
$$A_{80x100x50} \longrightarrow A'_{1x80x100}$$

$$A_{1x100x1} \longrightarrow A'_{1x1x100}$$



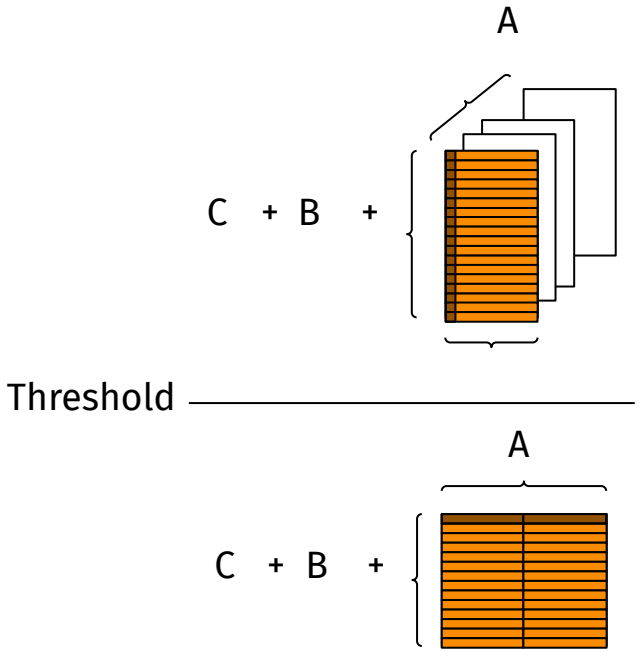
Phase 3 - TLB Miss Reduction

1 TLB entry = 50 elements



Phase 3 - TLB Miss Reduction

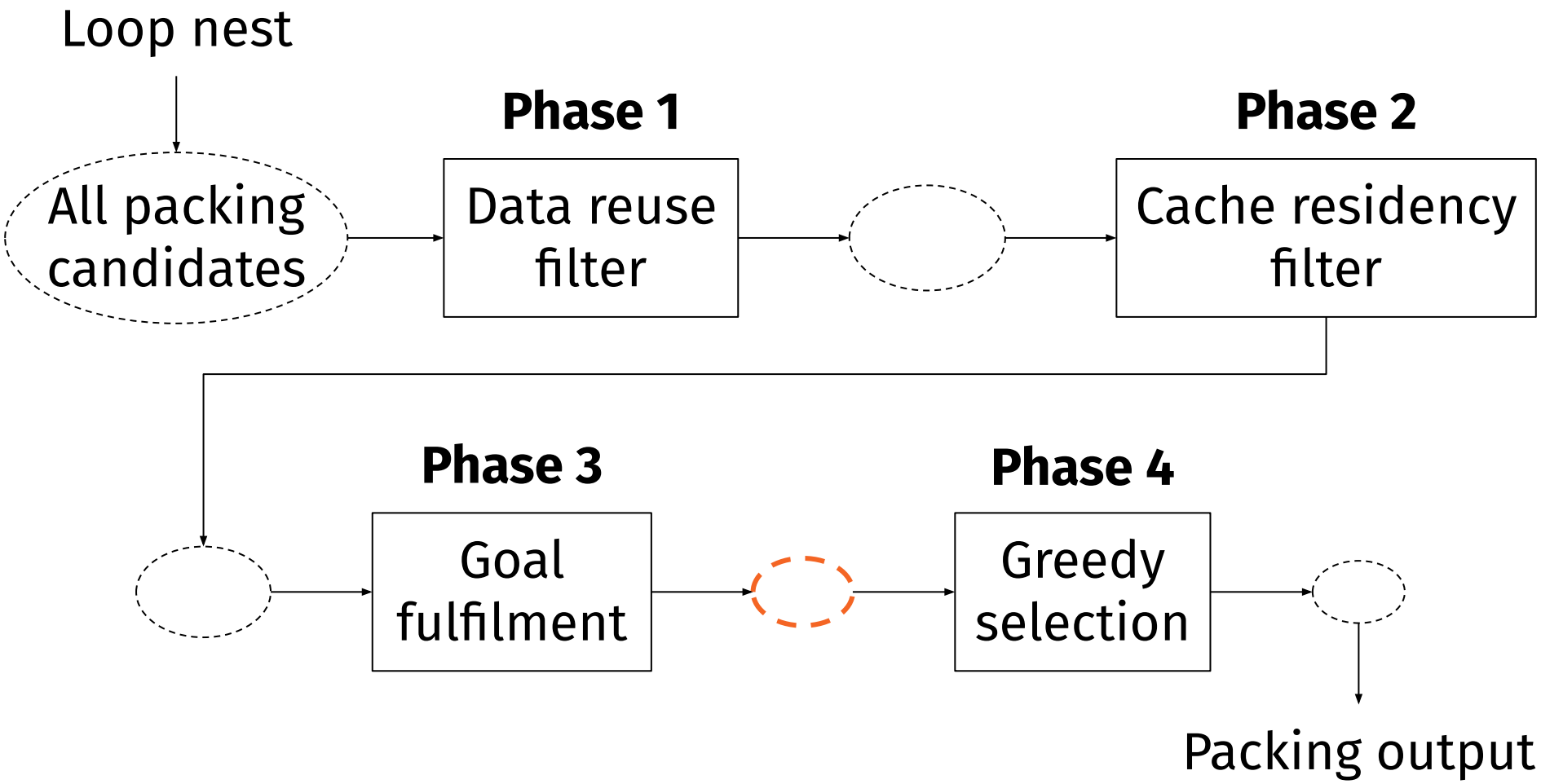
Packing reduces TLB entries
in a loop below L1 dTLB capacity?



Phase 3 - TLB Miss Reduction

Packing reduces TLB entries
in a loop below L1 dTLB capacity?

- For all **loops affected**
- For all **tensors accessed**
- Given packing and possible **data-layout change**


Loop nest
Phase 1
Phase 2
All packing candidates
Data reuse filter
Cache residency filter
Phase 3
Phase 4
Goal fulfilment
Greedy selection
Packing output

Phase 4 - Greedy Selection

Final packing selection:

1. Sort candidates based on cost-benefit
2. Greedy selection

GPAT in MLIR Affine

```
➜ mlir-opt -affine-loop-pack input.mlir
```

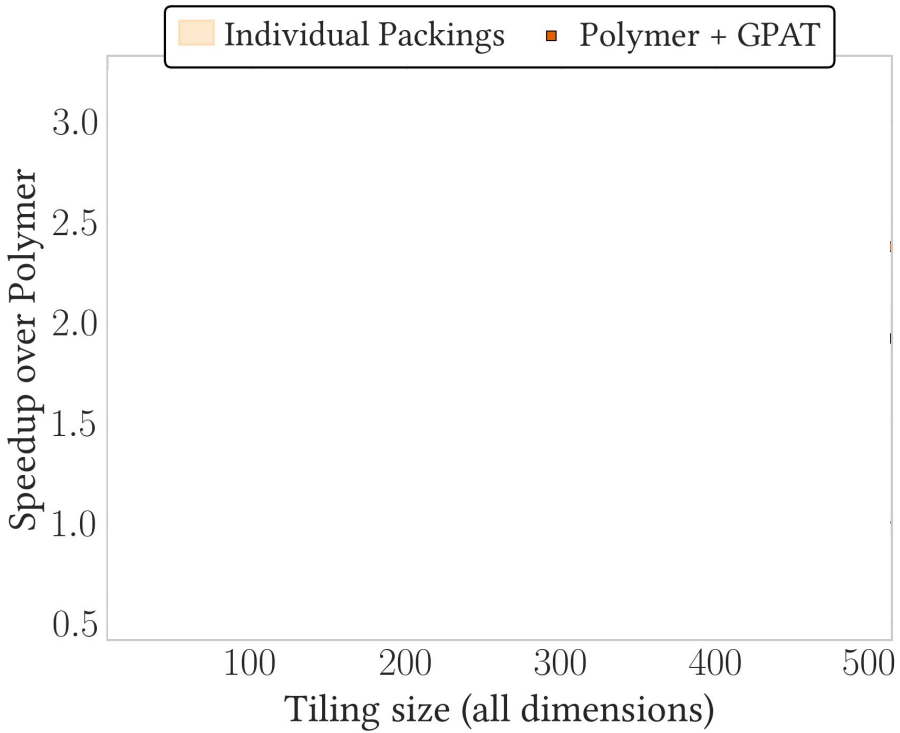
Artifact Available

https://doi.org/10.5281/zenodo.7517506

Evaluation

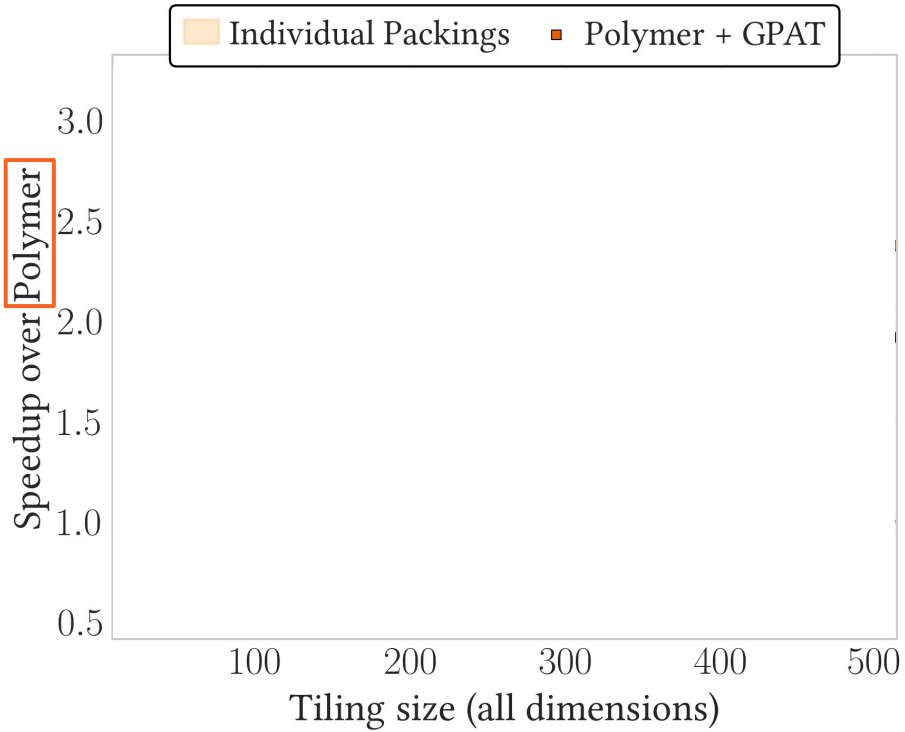
Packing Choice Evaluation

- Effective combination of candidates?



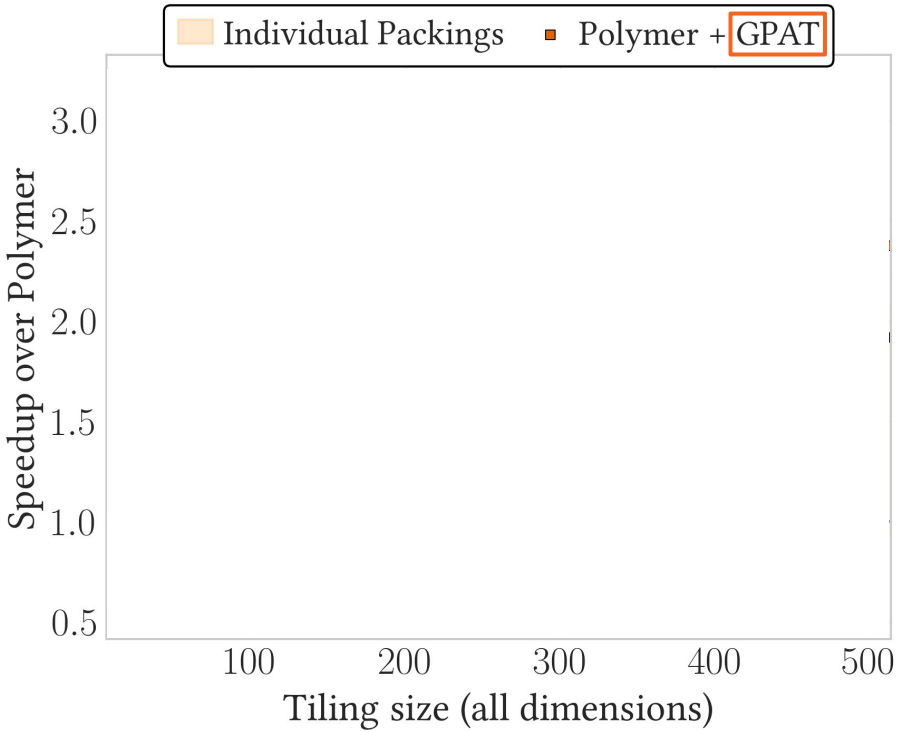
Packing Choice Evaluation

- Effective combination of candidates?



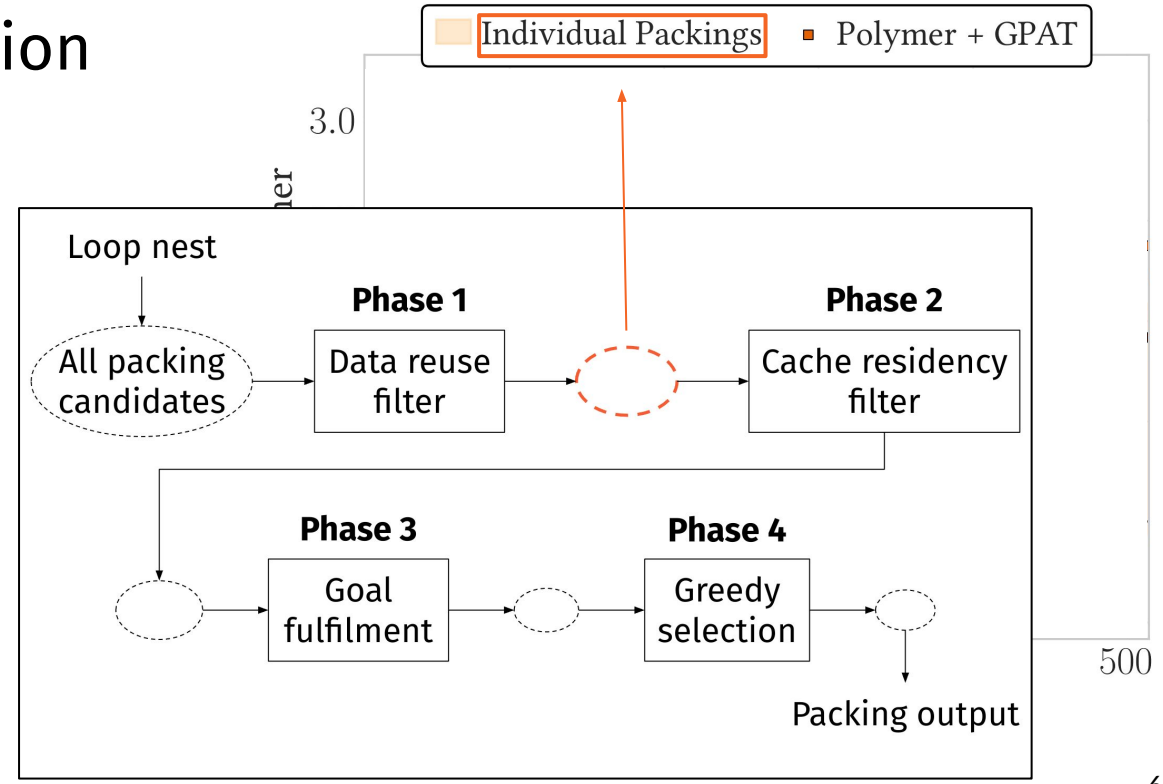
Packing Choice Evaluation

- Effective combination of candidates?



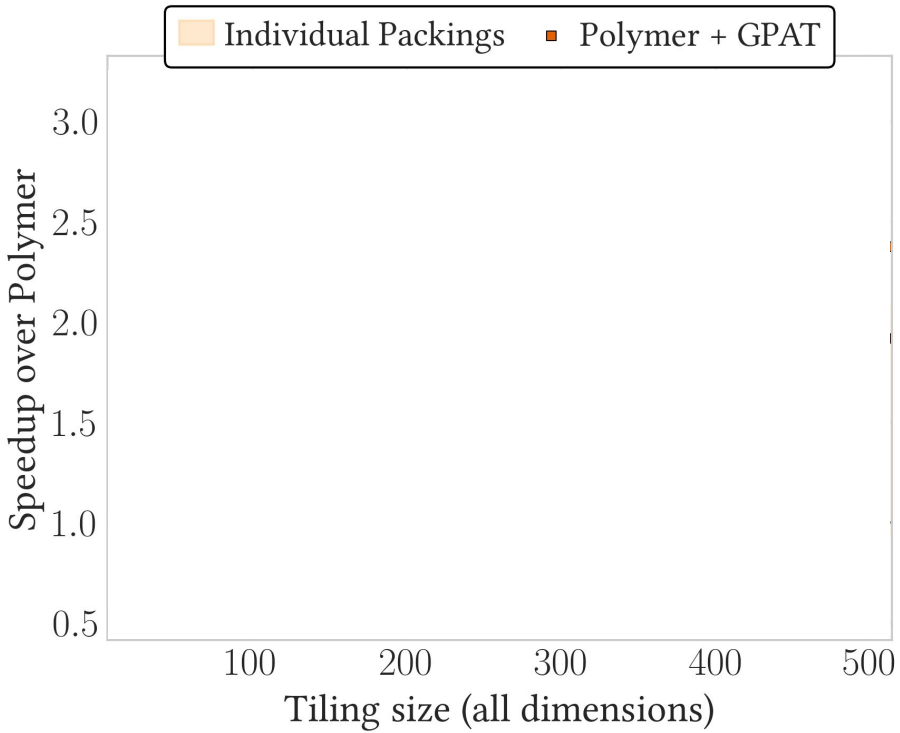
Packing Choice Evaluation

- Effective combination of candidates?

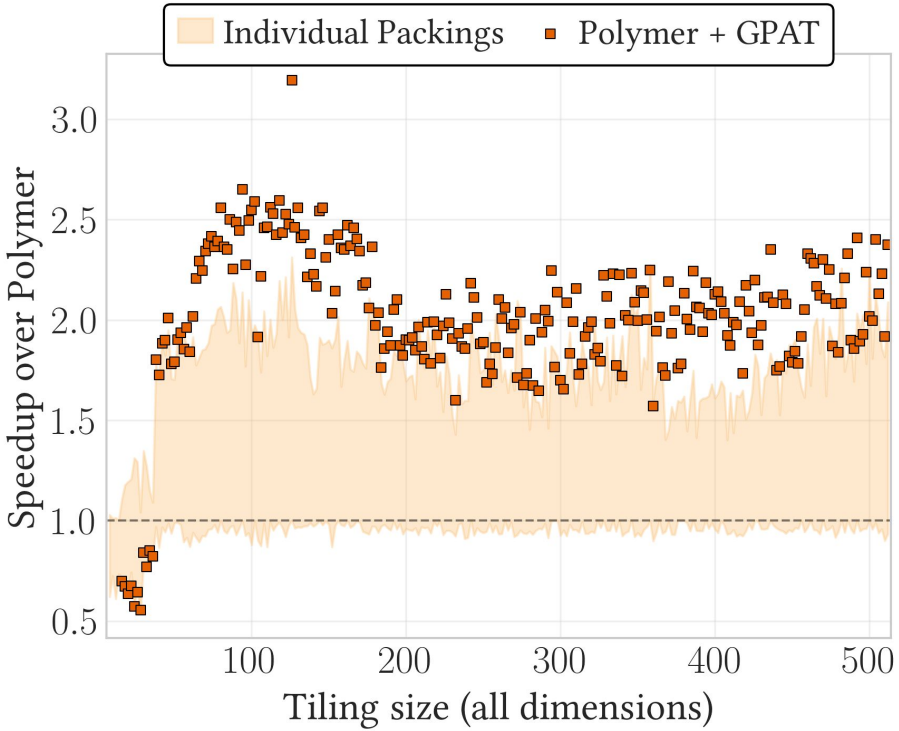


Packing Choice Evaluation

- Effective combination of candidates?



Packing Choice Evaluation - *2mm*

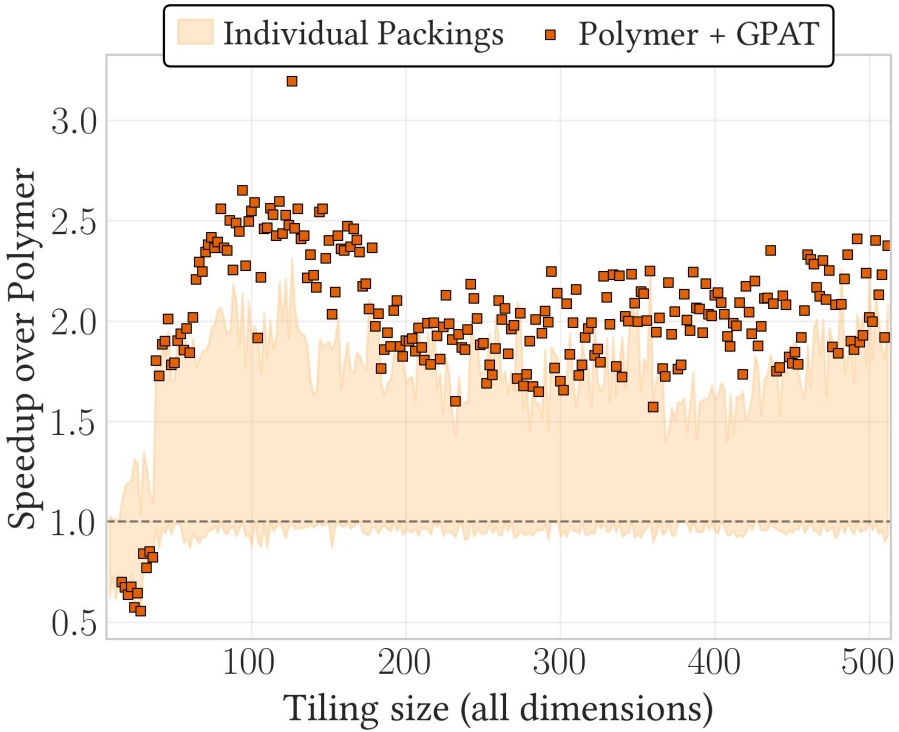


Packing Choice Evaluation - *2mm*

$$D_{M\times N} = \beta * D_{M\times N} + T_{M\times N} * C_{N\times N}$$

where

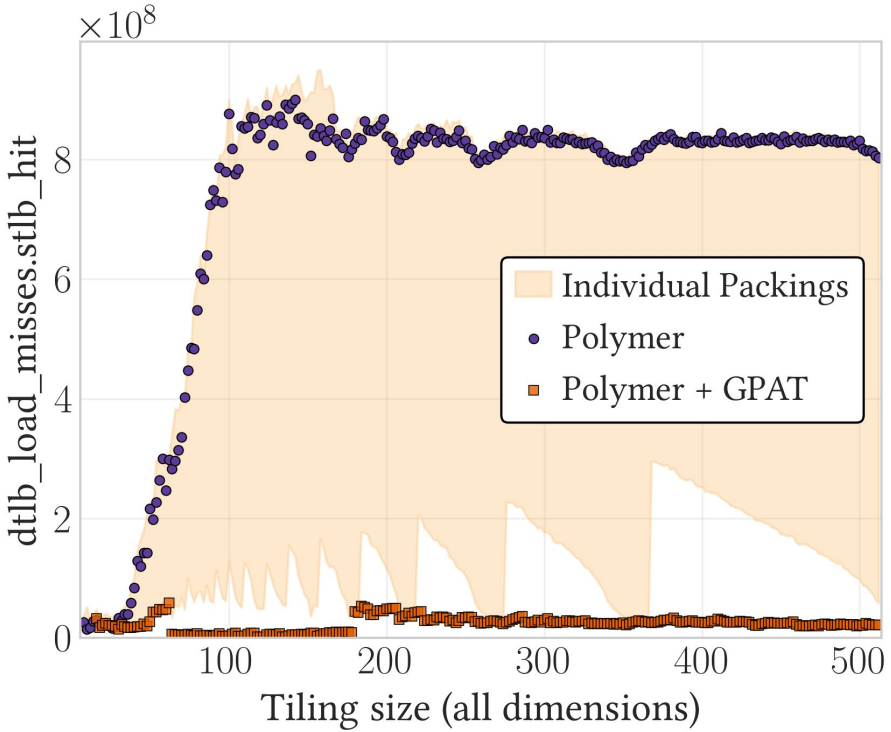
$$T_{M\times N} = \alpha * A_{M\times K} * B_{K\times N}$$



Packing Choice Evaluation - *2mm*

- L1 dTLB: miss
- L2 TLB: hit

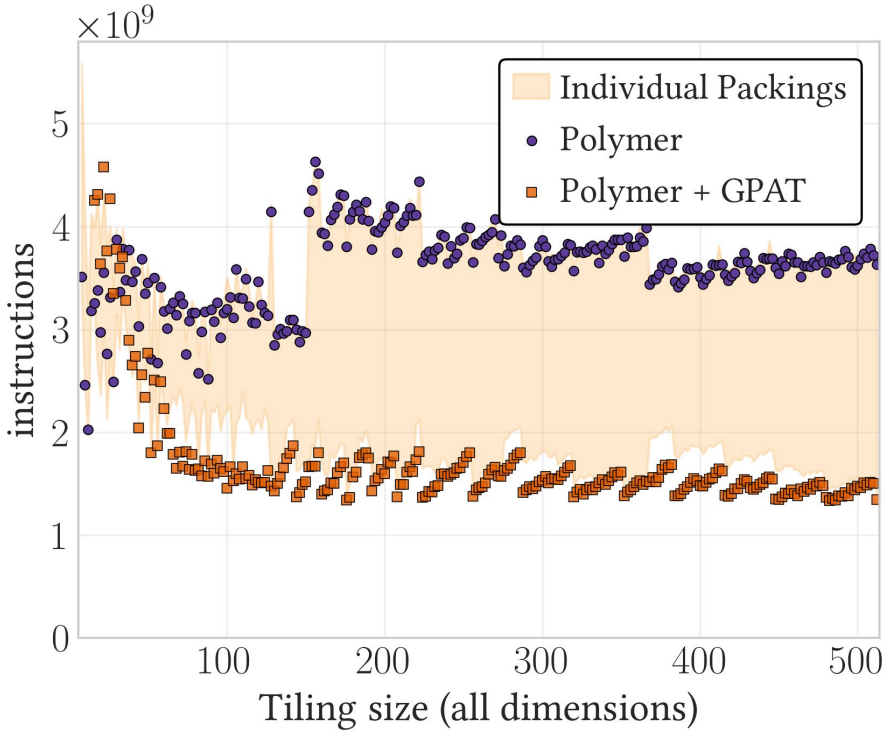
(lower is better)



Packing Choice Evaluation - *2mm*

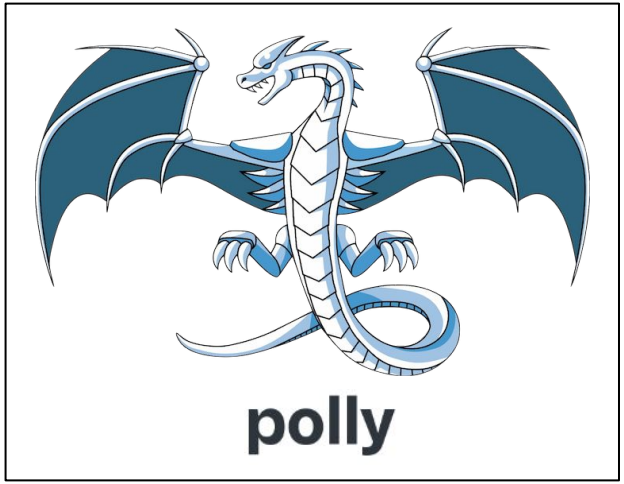
- Instruction count

(lower is better)



Polybench Evaluation

- GPAT compared to other approaches?
- Beyond *gemm*?
- Robust to prior transformations?



Polybench Evaluation

- GPAT compared to other approaches?
- Beyond *gemm*?
- Robust to prior transformations?

All polybench benchmarks

Polybench Evaluation

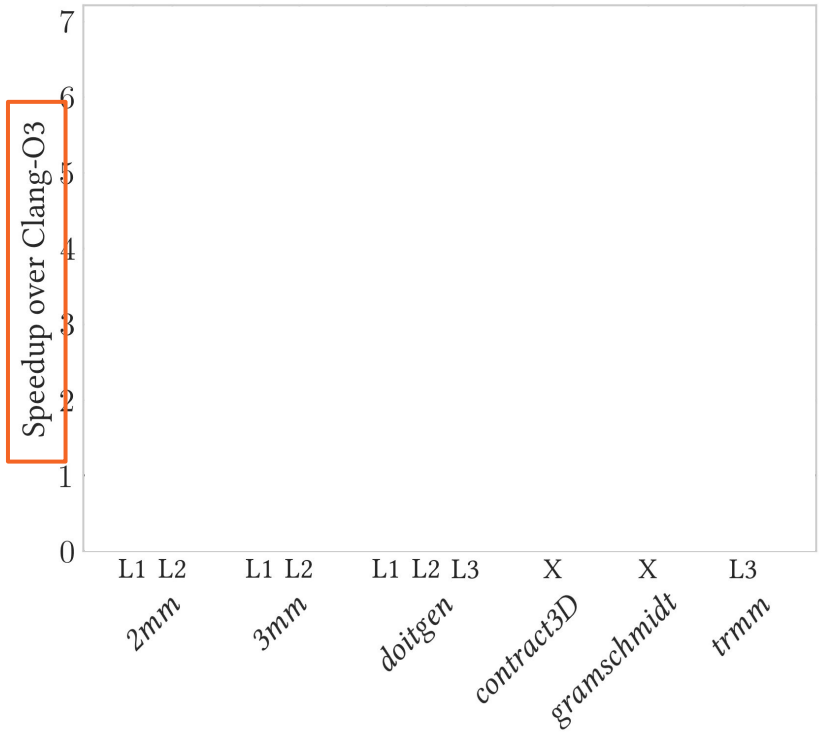
- GPAT compared to other approaches?
- Beyond *gemm*?
- Robust to prior transformations?

Tiling engine:

- Affine tiling

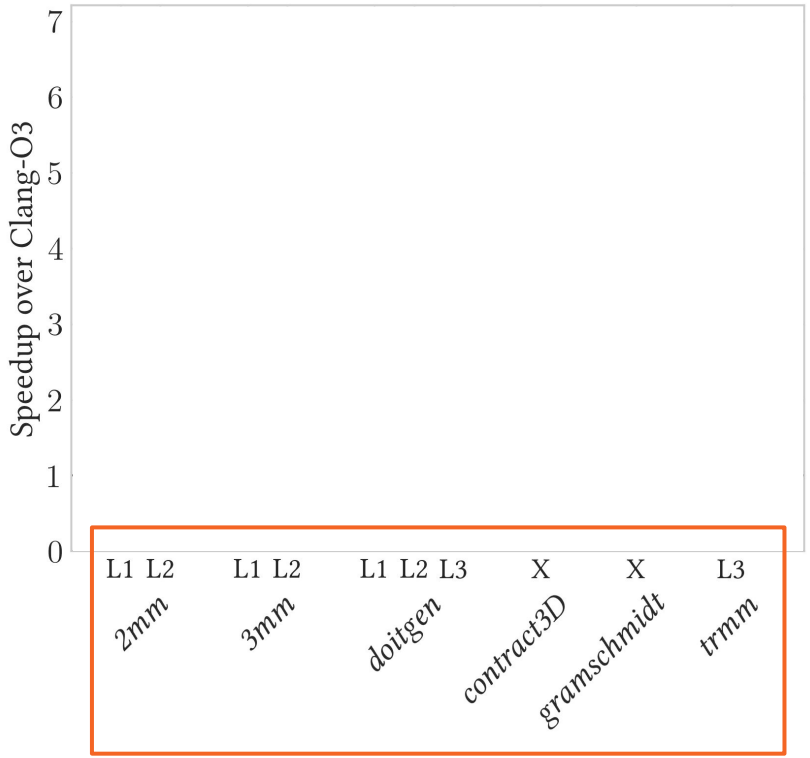
Polybench Evaluation - Affine Tiling

- Speedup over Clang -O3
- Benchmark + Tiling target
- Only benchmarks that were packed

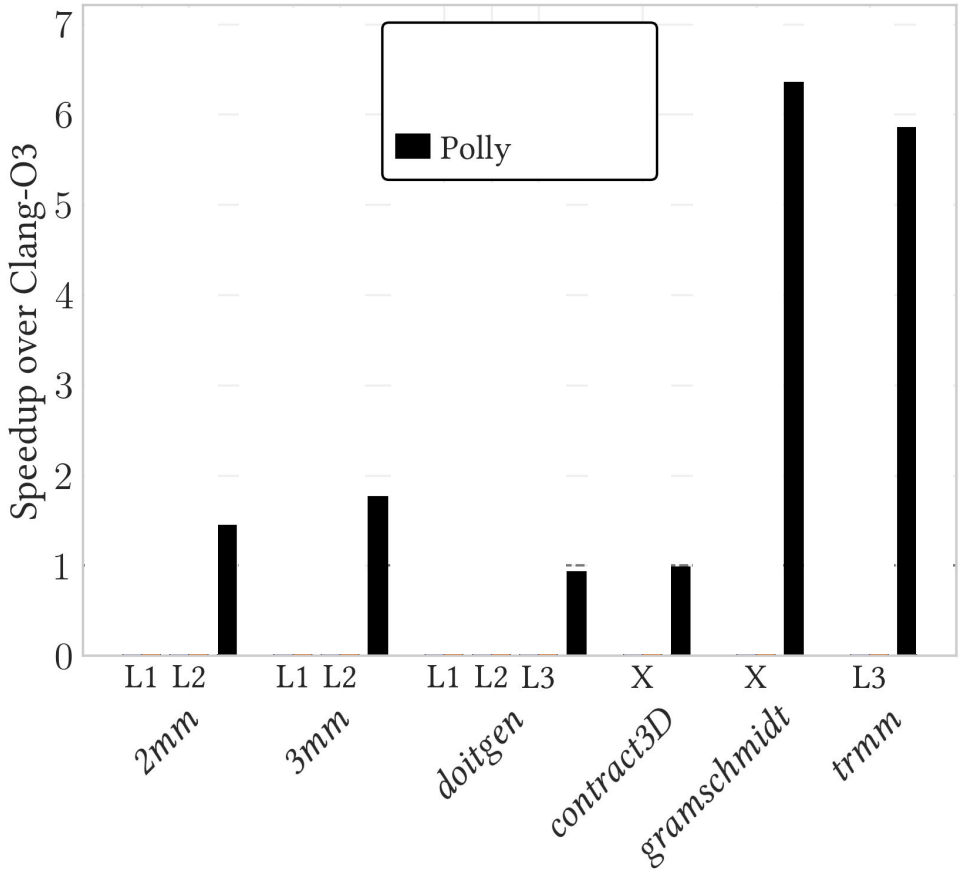


Polybench Evaluation - Affine Tiling

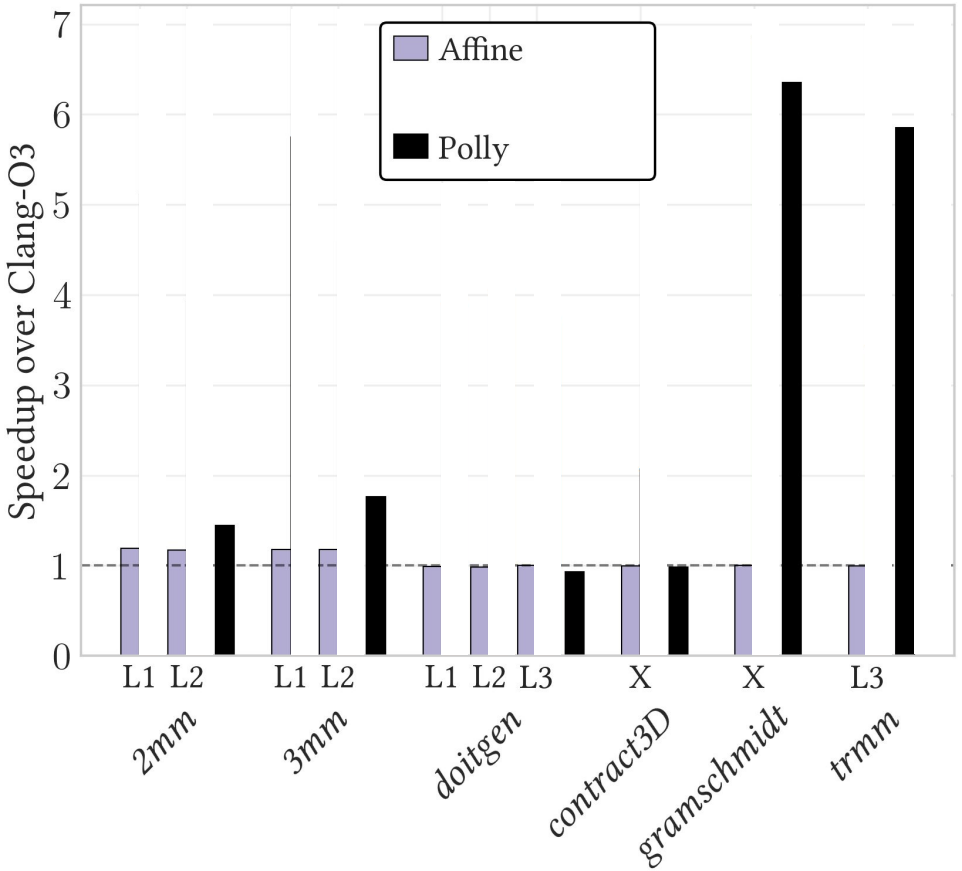
- Speedup over Clang -O3
- Benchmark + Tiling target
- Only benchmarks that were packed



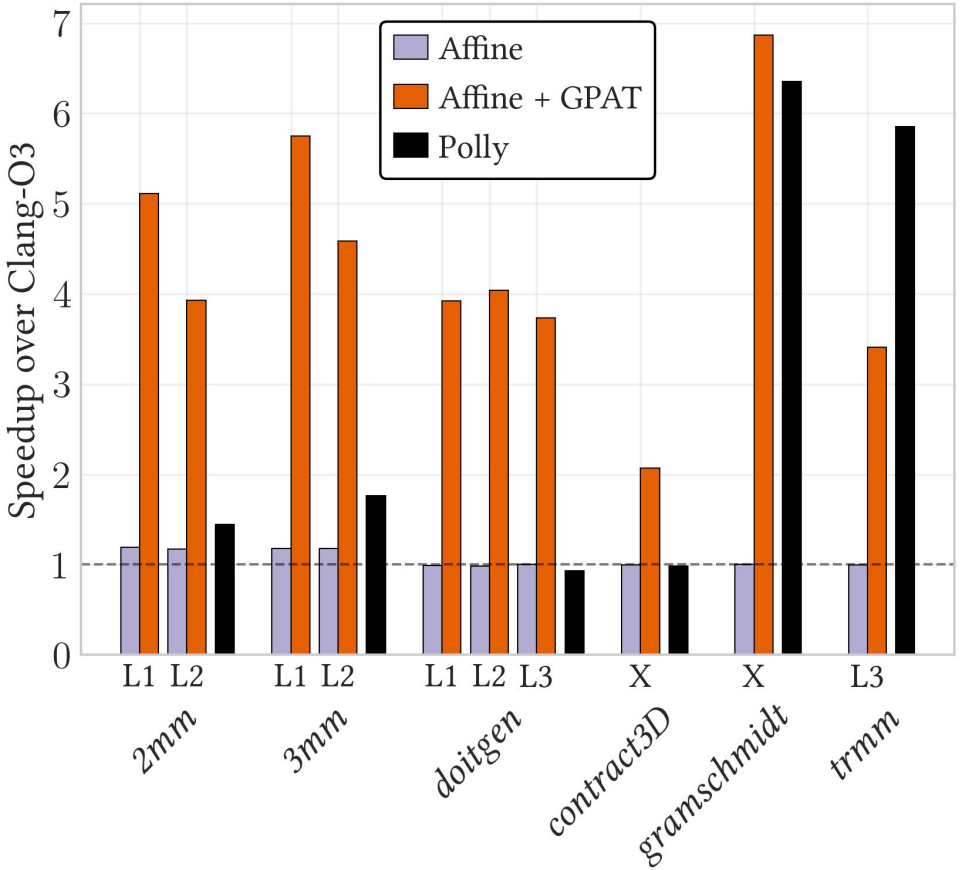
Polybench Evaluation - Affine Tiling



Polybench Evaluation - Affine Tiling

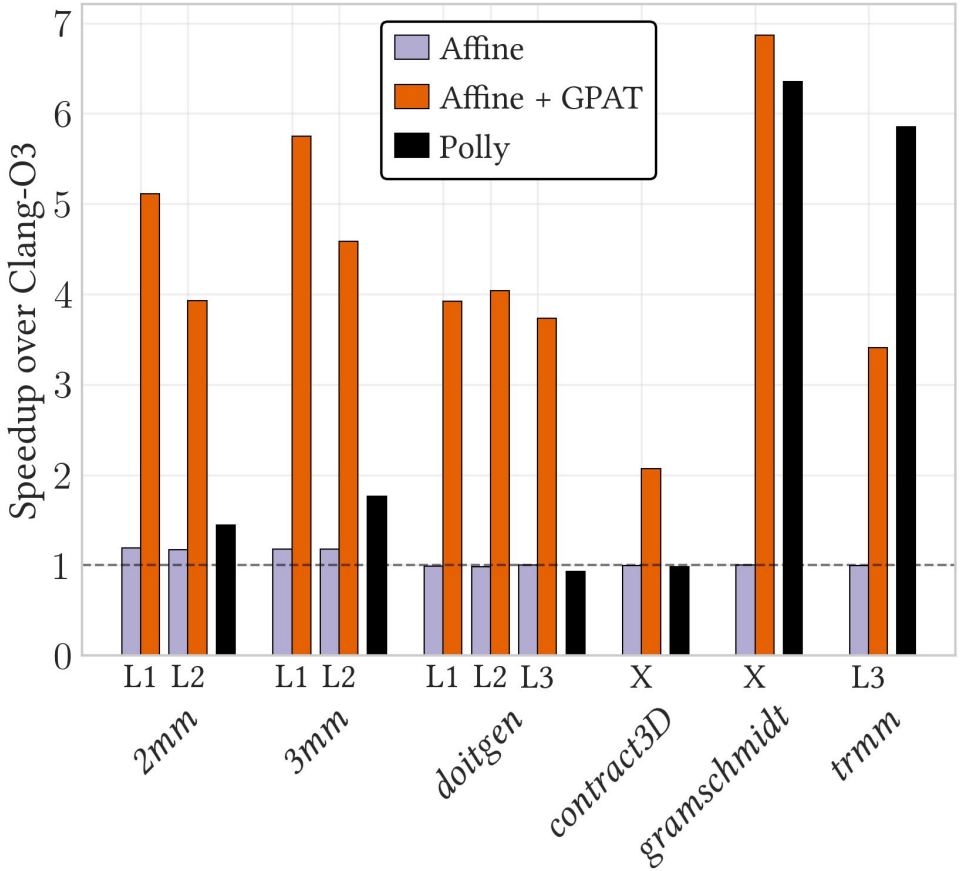


Polybench Evaluation - Affine Tiling



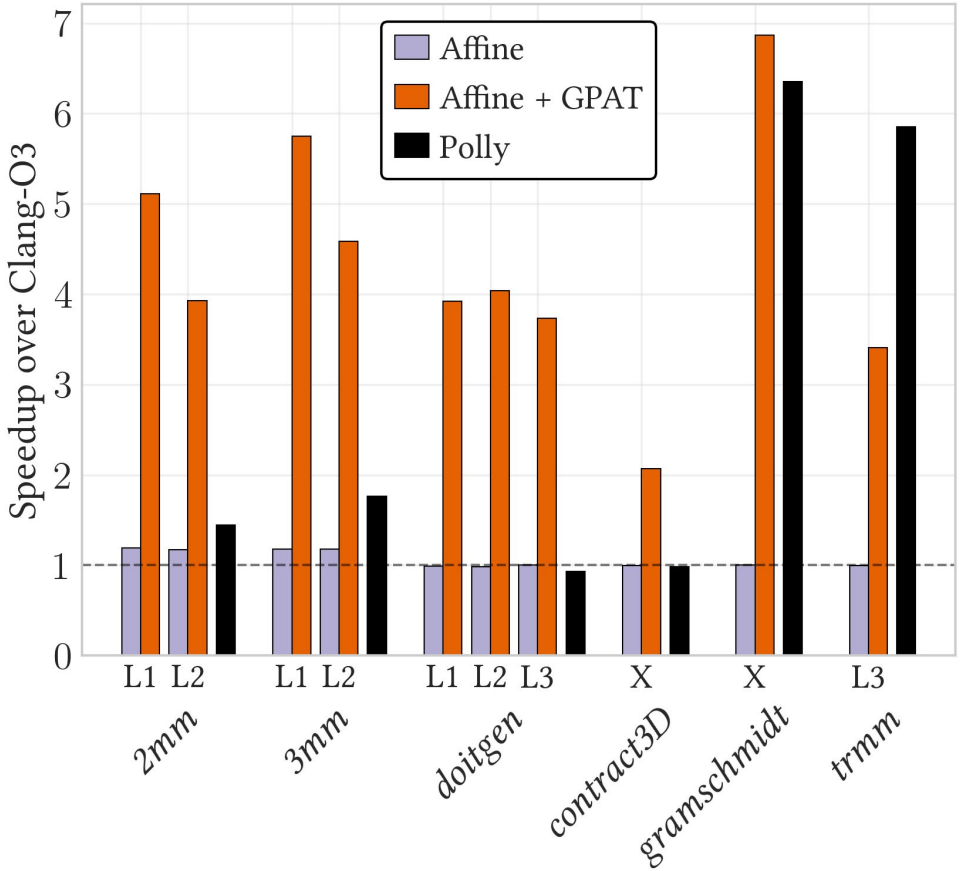
Polybench Evaluation - Affine Tiling

- Polybench is designed to evaluate polly



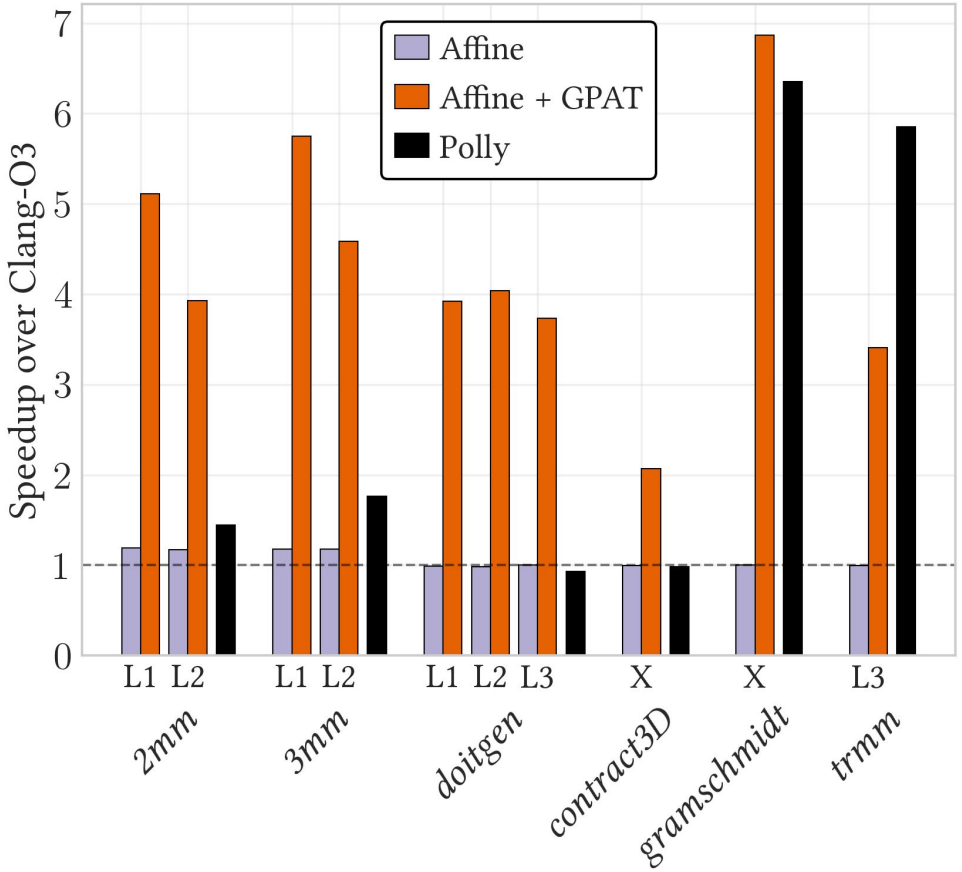
Polybench Evaluation - Affine Tiling

- Affine tiling is very simplistic



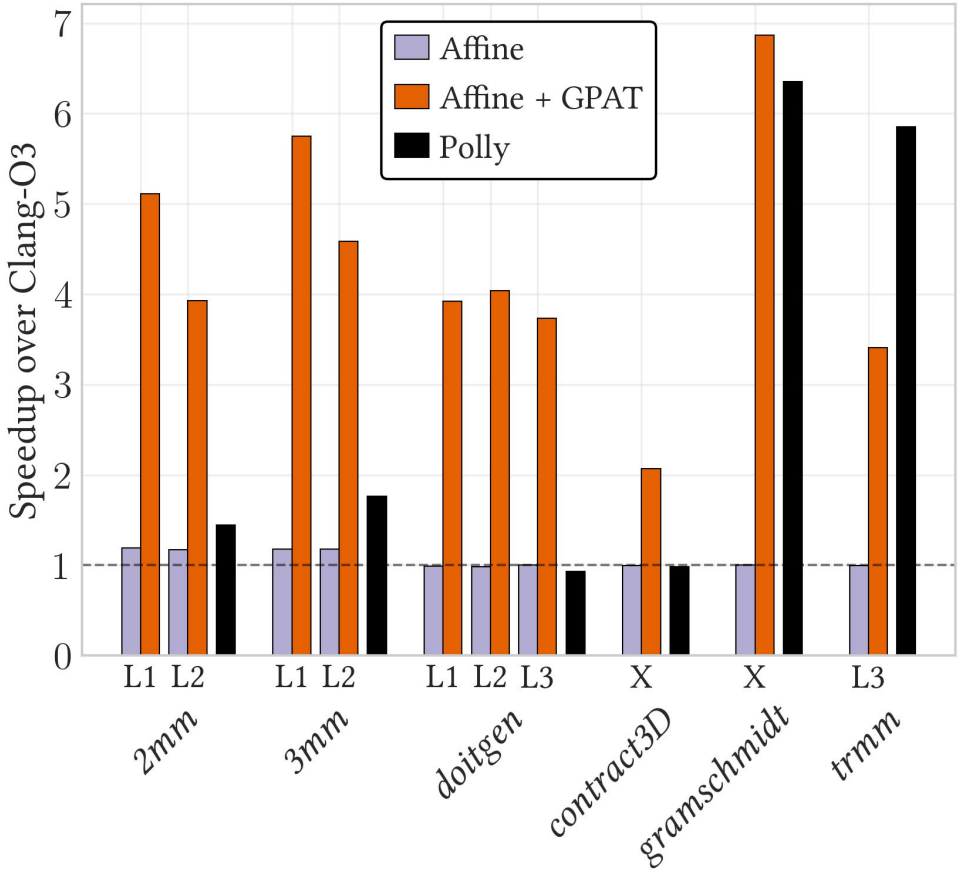
Polybench Evaluation - Affine Tiling

- Even when no tiling “X”



Polybench Evaluation - Affine Tiling

- GPAT improves GEMM-like and GEMM-unalike computations



To Pack or Not to Pack: A Generalized Packing Analysis and Transformation

Caio Salvador Rohwedder
csalvado@ualberta.ca

- Implemented in MLIR
- Evaluated in Polybench against Polly and Pluto

More Slides

Phase 1 - Data Reuse Filter

Candidate (**ForJ, A**):

- All accesses of **A** must be invariant to **j**

```
for(i=0; i<50; i++)
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A[k][l][i]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

Phase 1 - Data Reuse Filter

Candidate (**ForJ, A**):

- All accesses of **A** must be invariant to **j**

```
for(i=0; i<50; i++)
  for(j=0; j<60; j++)
    for(k=0; k<80; k++)
      for(l=0; l<100; l++)
        a = load A[k][l][i]
        b = load B[l][k][j]
        prod = mul a, b
        c = load C[i][j]
        sum = add c, prod
        store sum, C[i][j]
```

Phase 1 - Data Reuse Filter

Candidate (**ForJ, A**):

- All accesses of **A** must be invariant to **j**

```
for(i=0; i<50; i++)
  for(j=0; j<60; j++)
    for(k=0; k<80; k++)
      for(l=0; l<100; l++)
        a = load A[k][l][i]
        b = load B[l][k][j]
        prod = mul a, b
        c = load C[i][j]
        sum = add c, prod
        store sum, C[i][j]
```

Phase 1 - Data Reuse Filter

Candidate (**L**, **T**):

- All accesses of **T** must be invariant to the **IV of L**

Phase 1 - Data Reuse Filter

Candidate (**L**, **T**):

- All accesses of **T**
  must be invariant
  to the **IV of L**,
  and any IVs that
  depend on the IV of L

Phase 2 - Cache Residency Filter

Ensure A’ remains resident:

- Footprint of A’
- Footprint of working set of B and C in one iteration of ForJ

```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A’[0][k][l]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

Phase 2 - Cache Residency Filter

Ensure A’ remains resident:

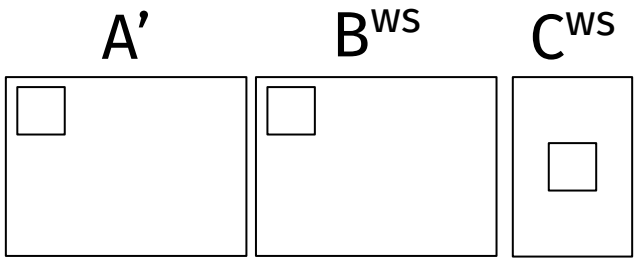
- Footprint of A’
- Footprint of working set of B and C in one iteration of ForJ

```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A’[0][k][l]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

Phase 2 - Cache Residency Filter

Ensure A’ remains resident:

- Iter j=0

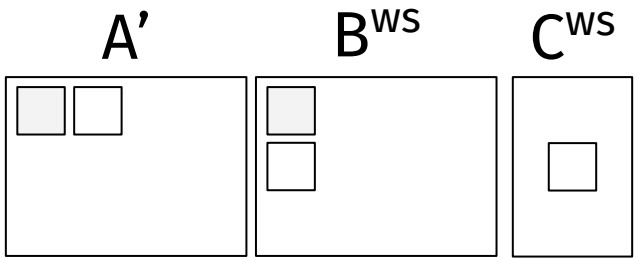



```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A’[0][k][l]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

Phase 2 - Cache Residency Filter

Ensure A’ remains resident:

- Iter j=0

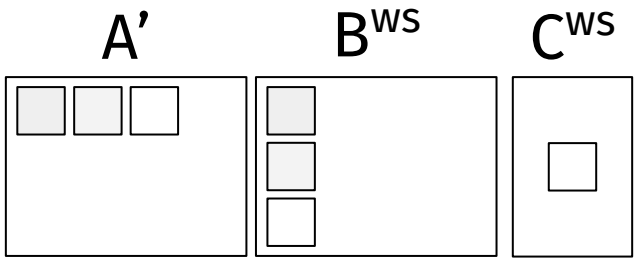



```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A’[0][k][l]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

Phase 2 - Cache Residency Filter

Ensure A’ remains resident:

- Iter j=0

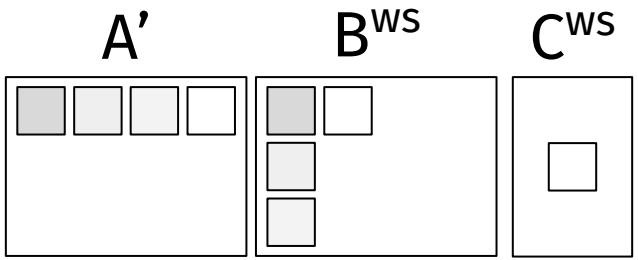



```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A’[0][k][l]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

Phase 2 - Cache Residency Filter

Ensure A’ remains resident:

- Iter j=0

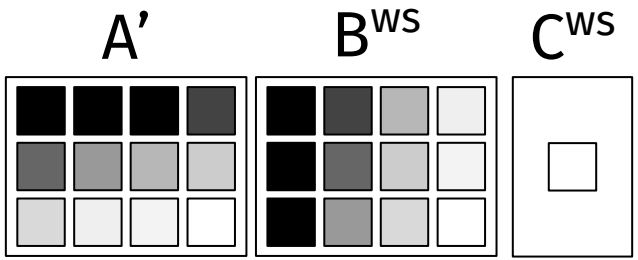



```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A’[0][k][l]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

Phase 2 - Cache Residency Filter

Ensure A’ remains resident:

- Iter j=0

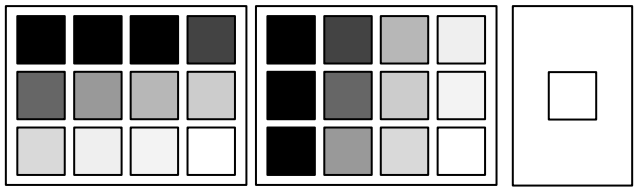



```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A’[0][k][l]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

Phase 2 - Cache Residency Filter

Ensure A’ remains resident:

- Iter j=1



```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A’[0][k][l]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

Phase 2 - Cache Residency Filter

Ensure A’ remains resident:

- Footprint of A’
- **Twice** the footprint of working set of B and C in one iteration of ForJ

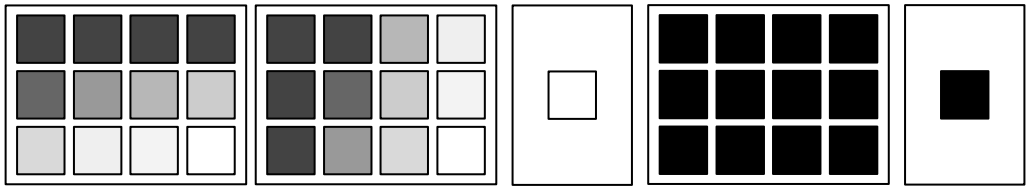
```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A’[0][k][l]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

Phase 2 - Cache Residency Filter

Ensure A’ remains resident:

- Footprint of A’
- **Twice** the footprint of working set of B and C in one iteration of ForJ

A’

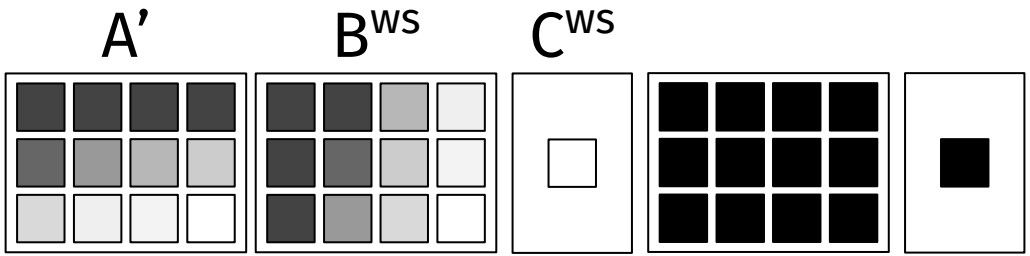


```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A’[0][k][l]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

Phase 2 - Cache Residency Filter

Ensure A’ remains resident:

- Footprint of A’
- **Twice** the footprint of working set of B and C in one iteration of ForJ

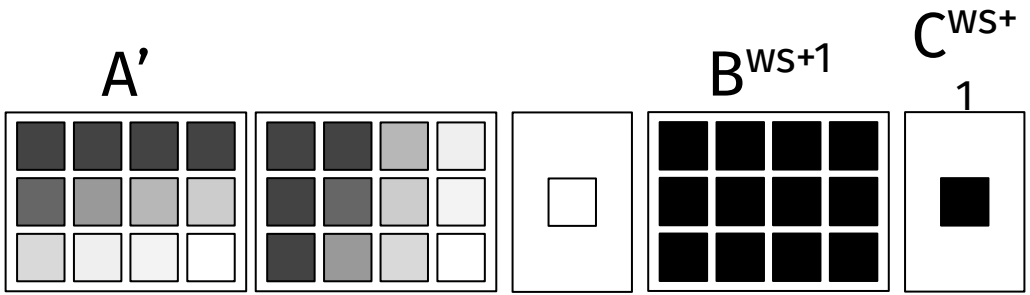



```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
for(j=0; j<60; j++)
 for(k=0; k<80; k++)
  for(l=0; l<100; l++)
   a = load A’[0][k][l]
   b = load B[l][k][j]
   prod = mul a, b
   c = load C[i][j]
   sum = add c, prod
   store sum, C[i][j]
```

Phase 2 - Cache Residency Filter

Ensure A’ remains resident:

- Footprint of A’
- **Twice** the footprint of working set of B and C in one iteration of ForJ

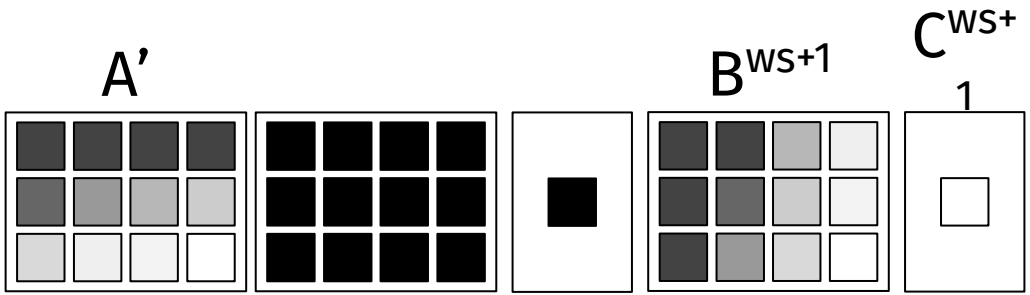



```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A’[0][k][l]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

Phase 2 - Cache Residency Filter

Ensure A’ remains resident:

- Footprint of A’
- **Twice** the footprint of working set of B and C in one iteration of ForJ

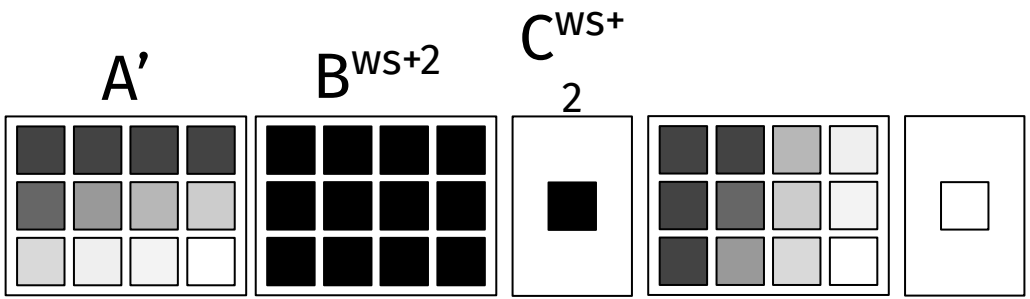



```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
 for(j=0; j<60; j++)
  for(k=0; k<80; k++)
   for(l=0; l<100; l++)
    a = load A’[0][k][l]
    b = load B[l][k][j]
    prod = mul a, b
    c = load C[i][j]
    sum = add c, prod
    store sum, C[i][j]
```

Phase 2 - Cache Residency Filter

Ensure A’ remains resident:

- Footprint of A’
- **Twice** the footprint of working set of B and C in one iteration of ForJ

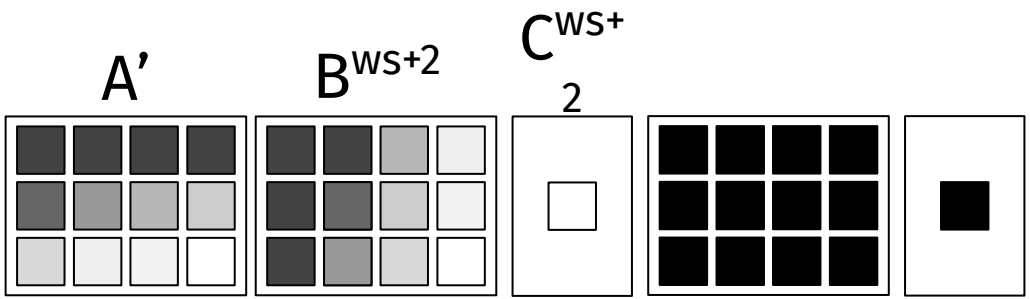



```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
for(j=0; j<60; j++)
 for(k=0; k<80; k++)
  for(l=0; l<100; l++)
   a = load A’[0][k][l]
   b = load B[l][k][j]
   prod = mul a, b
   c = load C[i][j]
   sum = add c, prod
   store sum, C[i][j]
```

Phase 2 - Cache Residency Filter

Ensure A’ remains resident:

- Footprint of A’
- **Twice** the footprint of working set of B and C in one iteration of ForJ




```
for(i=0; i<50; i++)
 A’ = alloc(1x80x100)
 for(m=0; m<80; m++)
  for(n=0; n<100; n++)
   tmp = load A[m][n][i]
   store tmp, A’[0][m][n]
for(j=0; j<60; j++)
 for(k=0; k<80; k++)
  for(l=0; l<100; l++)
   a = load A’[0][k][l]
   b = load B[l][k][j]
   prod = mul a, b
   c = load C[i][j]
   sum = add c, prod
   store sum, C[i][j]
```

Phase 2 - Cache Residency Filter

Ensure T’ remains resident:

- Footprint of T’
- Twice the footprint of working set of all tensors in one iteration of L

Phase 4 - Greedy Selection

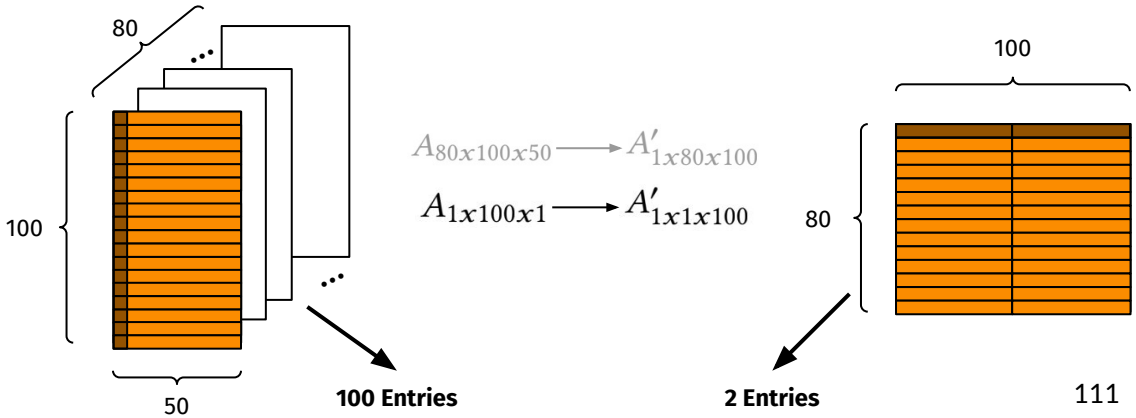
1. Sort cost-benefit of candidates:

$$\frac{\text{TLB Improvement}}{\text{Footprint of T'} \quad (x2)}$$

Phase 4 - Greedy Selection

1. Sort cost-benefit of candidates:

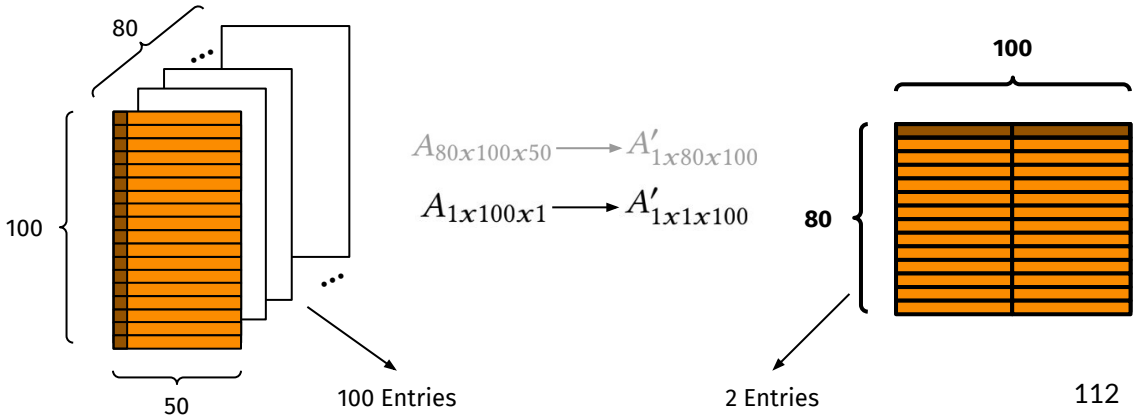
$$\frac{\textbf{TLB Improvement}}{\text{Footprint of T'}} \quad (x2)$$



Phase 4 - Greedy Selection

1. Sort cost-benefit of candidates:

$$\frac{\text{TLB Improvement}}{\textbf{Footprint of T'}} \quad (x2)$$



Phase 4 - Greedy Selection

2. Greedy selection, checking:
    - Redundant packings
    - Candidates benefit in presence of previous selection

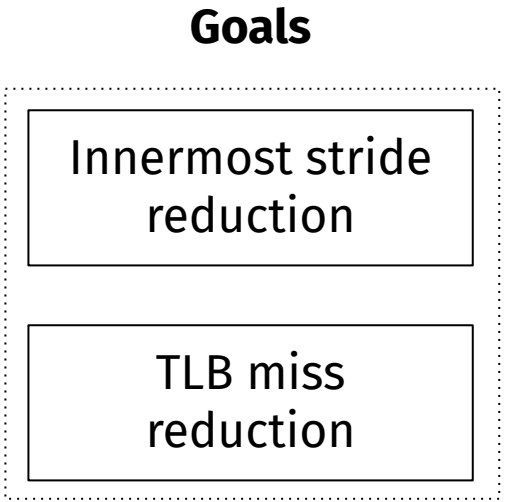
Phase 4 - Greedy Selection

2. Greedy selection, checking:
   - **Redundant packings**
   - Candidates benefit in presence of previous selection

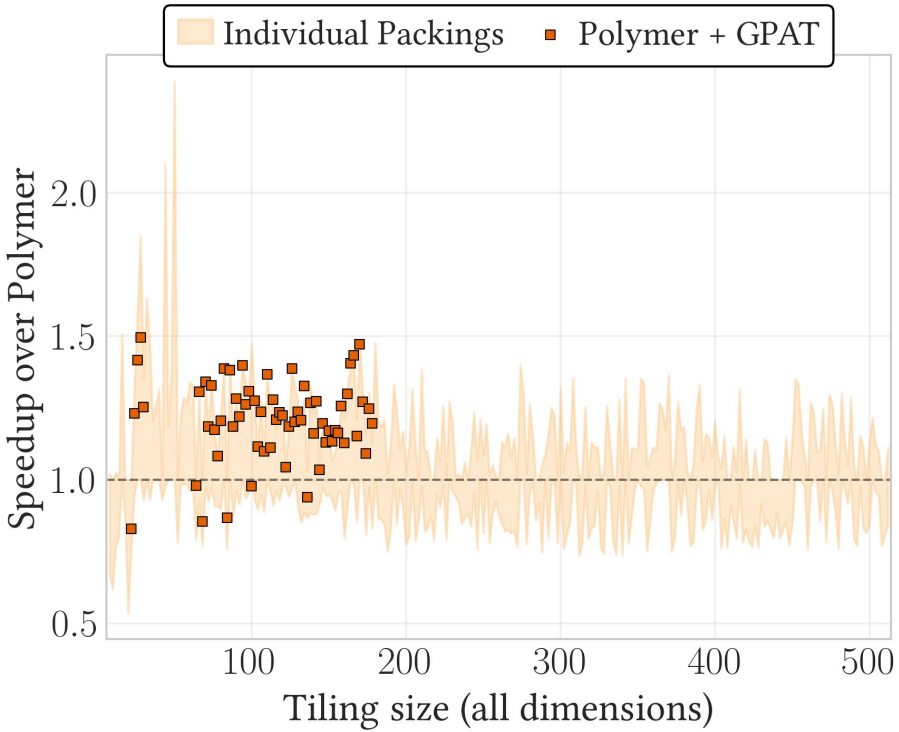
```
for(i=0; i<50; i++)
for(j=0; j<60; j++)
 for(k=0; k<80; k++)
  for(l=0; l<100; l++)
   a = load A[k][l][i]
   b = load B[l][k][j]
   prod = mul a, b
   c = load C[i][j]
   sum = add c, prod
   store sum, C[i][j]
```

Phase 4 - Greedy Selection

2. Greedy selection, checking:
   - Redundant packings
   - **Candidates benefit in presence of previous selection**

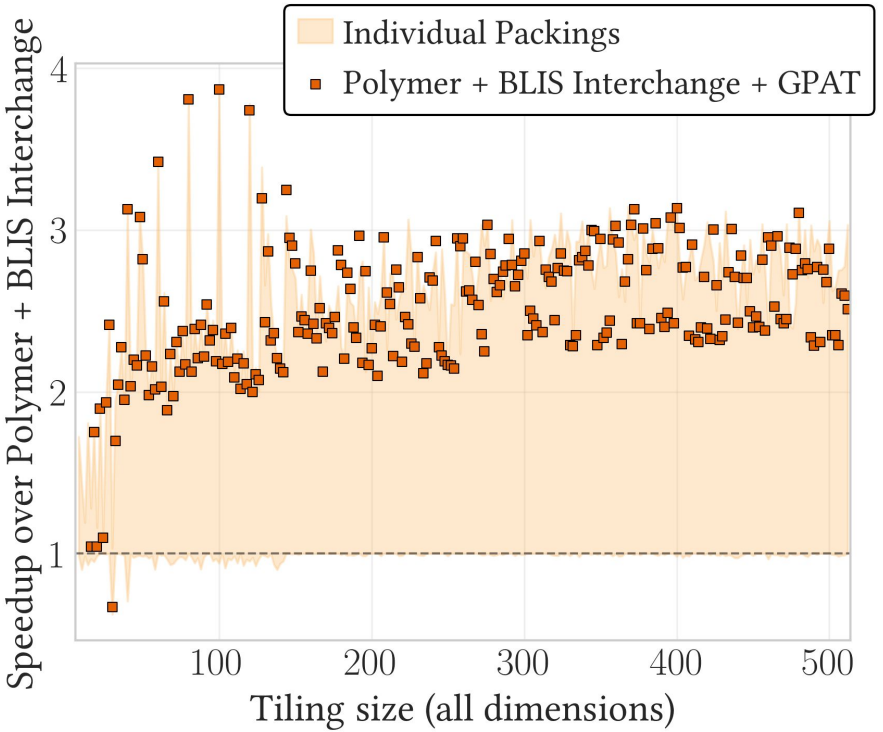


Packing Choice Evaluation - *gemm*



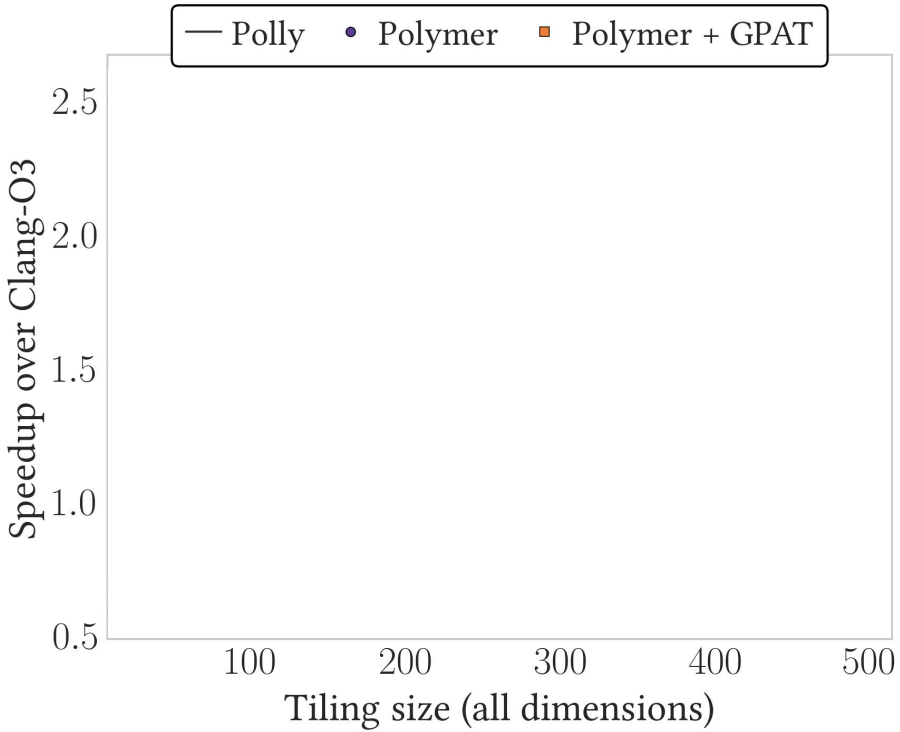
Packing Choice Evaluation - *gemm* BLIS

- *gemm* interchanged to BLIS loop ordering



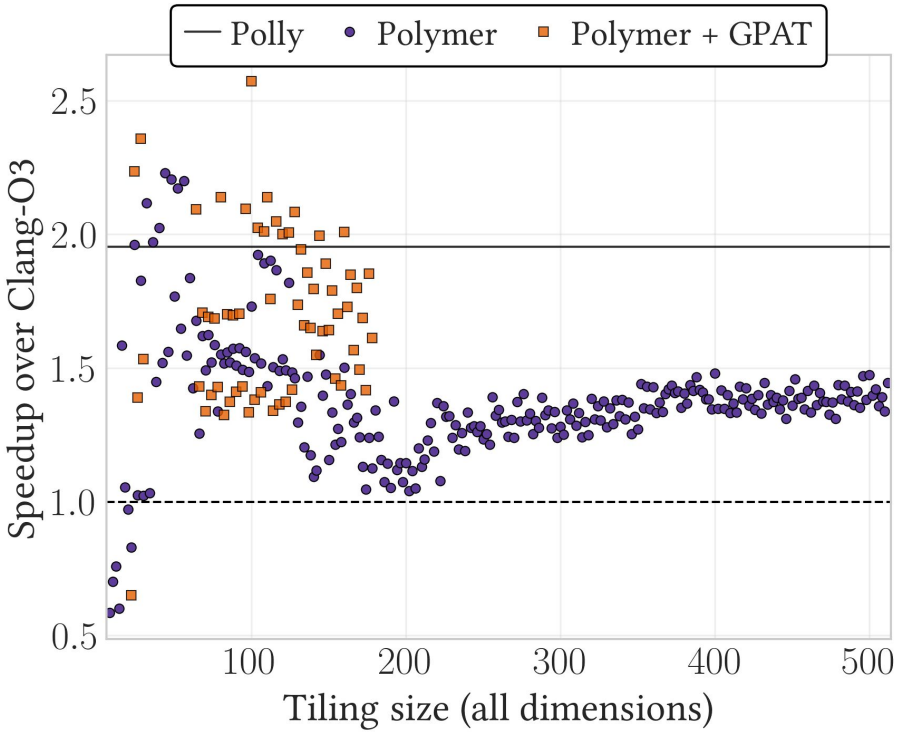
Polybench Evaluation - Polymer Tiling

- Speedup over Clang -O3
- Polly



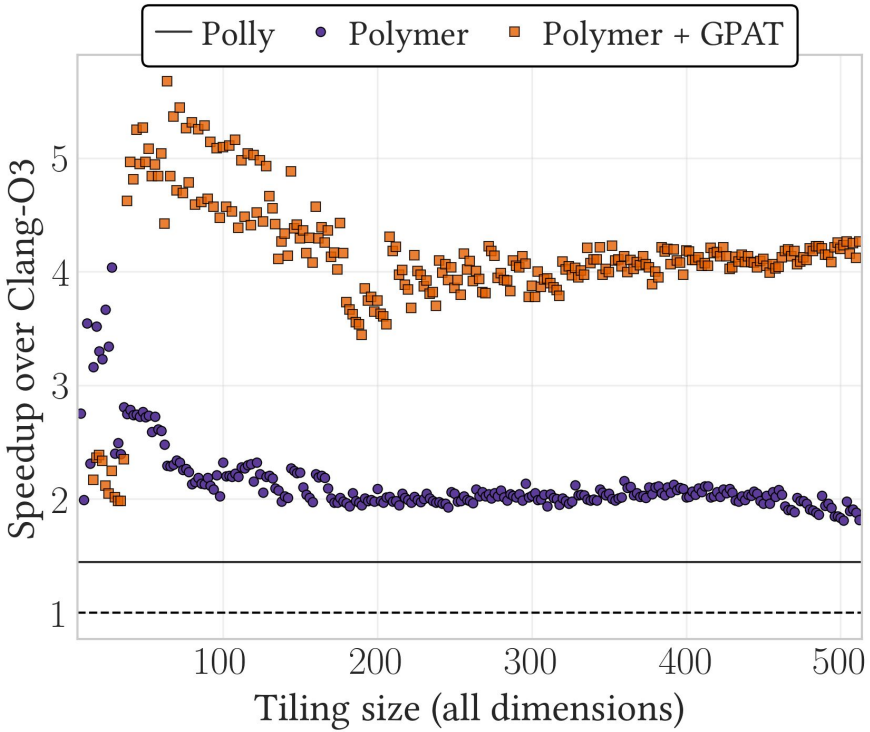
Polybench Evaluation - Polymer Tiling

gemm



Polybench Evaluation - Polymer Tiling

2mm



Polybench Evaluation - Polymer Tiling

3mm