Load Balancing with Polygonal Partitions

ANIKET SHIVAM

PRIYANKA RAVI

ALEXANDER V. VEIDENBAUM

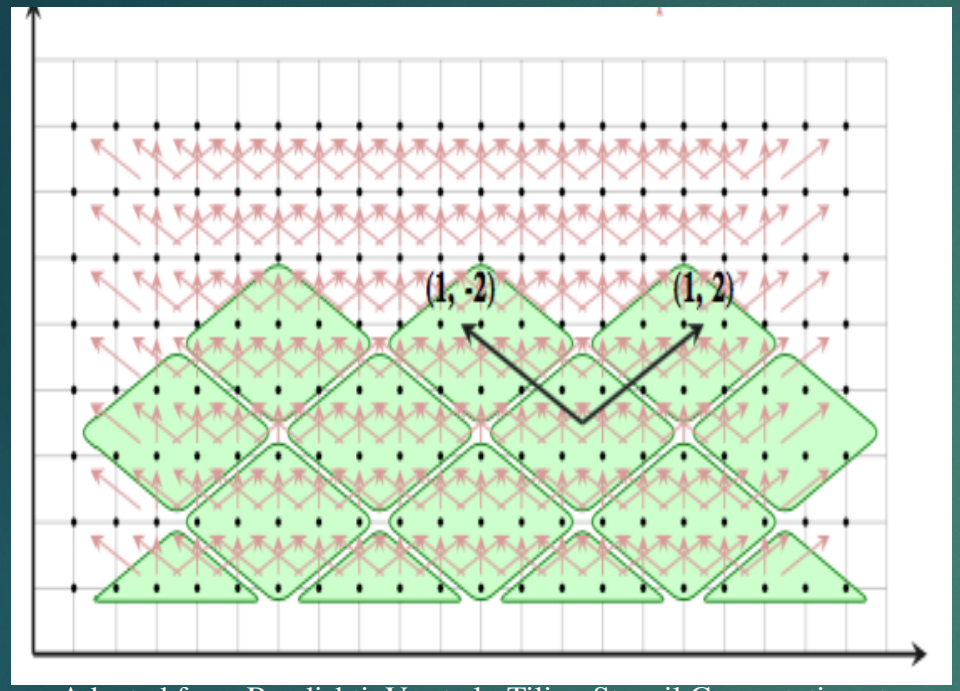
ALEXANDRU NICOLAU

UNIVERSITY OF CALIFORNIA, IRVINE, USA

ROSARIO CAMMAROTA

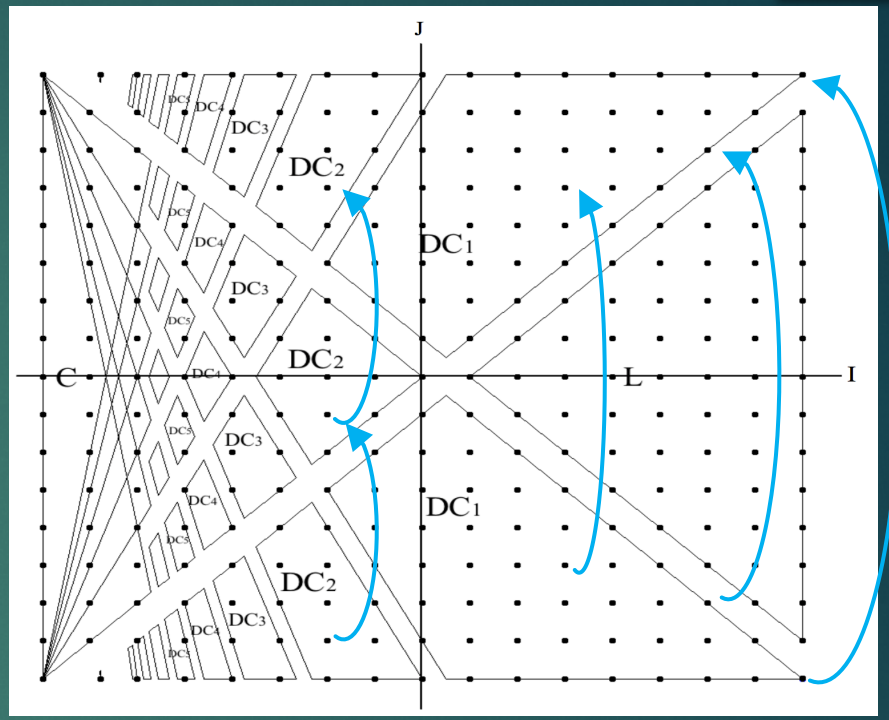
QUALCOMM RESEARCH, SAN DIEGO, USA

Traditional Tiling vs Polygonal Tiling



Adapted from Bandishti, V., et al.: Tiling Stencil Computations to Maximize Parallelism. In: SC '12.

- Single shape of tiles. (not necessarily the size)
- Improves data locality for loop-nests with uniform reuse pattern.



Varying reuse distances

- Multiple tile sizes and shapes based on reuse pattern.
- Improves data locality for loop-nests with non-uniform reuse pattern.

Formulation of the Problem

- **Walk-through example:**

```
for ( i = -N; i <= N; i++)
    for ( j = -N; j <= N; j++)
        X[i,j] = Y[i,i+j+3] + Y[i+j,j];
```

- **Representation of the references to characterize the reuse pattern**:

Reference $\alpha = (i, i+j+3)$

Reference $\beta = (i+j, j)$

$$\Gamma_{i,i+j+3} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} I + \begin{pmatrix} 0 \\ 3 \end{pmatrix} \text{ and } \Gamma_{i+j,j} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} I + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Formulation of the Problem

- **Deriving the other iteration reusing the same data:**

$$\Gamma_\alpha = \Gamma_\beta \quad \Leftrightarrow \quad \mathbf{R}_\alpha \mathbf{I}_\alpha + \mathbf{r}_\alpha = \mathbf{R}_\beta \mathbf{I}_\beta + \mathbf{r}_\beta$$

- **Temporal reuse relation:**

$$R_\beta^{-1} R_\alpha \mathbf{I}_\alpha + R_\beta^{-1}(\mathbf{r}_\alpha - \mathbf{r}_\beta) = \mathbf{I}_\beta \quad \Leftrightarrow \quad \mathbf{T}_{\alpha\beta}\mathbf{I}_\alpha + \mathbf{t}_{\alpha\beta} = \mathbf{I}_\beta$$

$$\mathcal{T} = (\mathrm{T}, \mathrm{t})$$

- **For the example**:

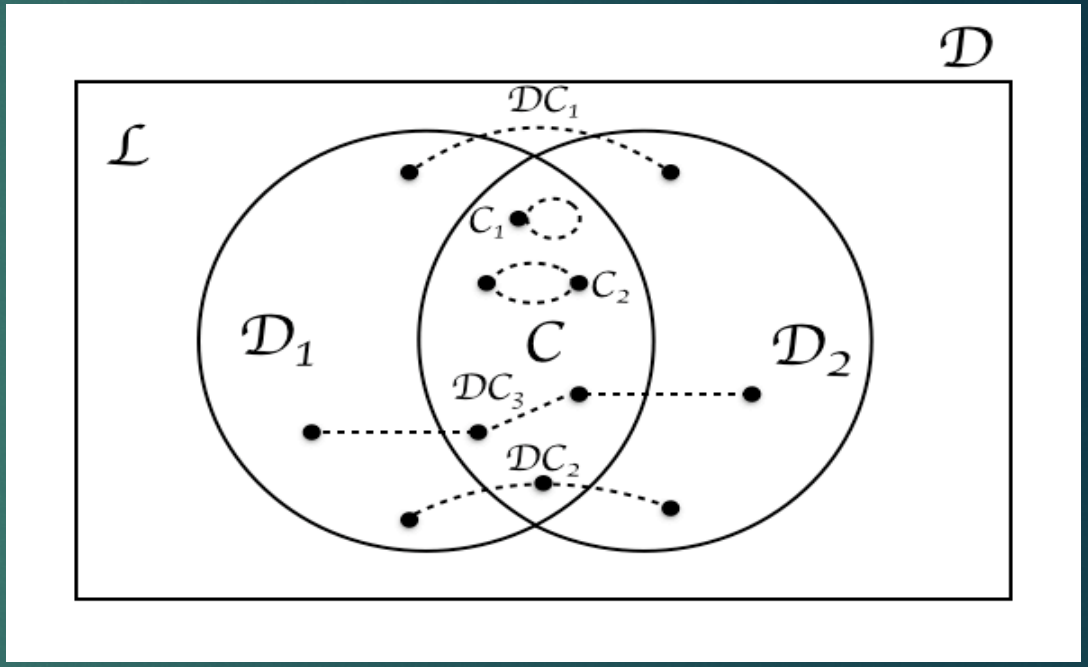
$$\mathrm{T} = \begin{pmatrix} 0 & -1 \\ 1 & 1 \end{pmatrix} \text{ and } \mathrm{t} = \begin{pmatrix} -3 \\ 3 \end{pmatrix}$$

- To find iteration reusing same data as **(2, 1)**:

$$\begin{pmatrix} 0 & -1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{2} \\ \mathbf{1} \end{pmatrix} + \begin{pmatrix} -3 \\ 3 \end{pmatrix} = \begin{pmatrix} \mathbf{-4} \\ \mathbf{6} \end{pmatrix}$$

Partitioning Technique

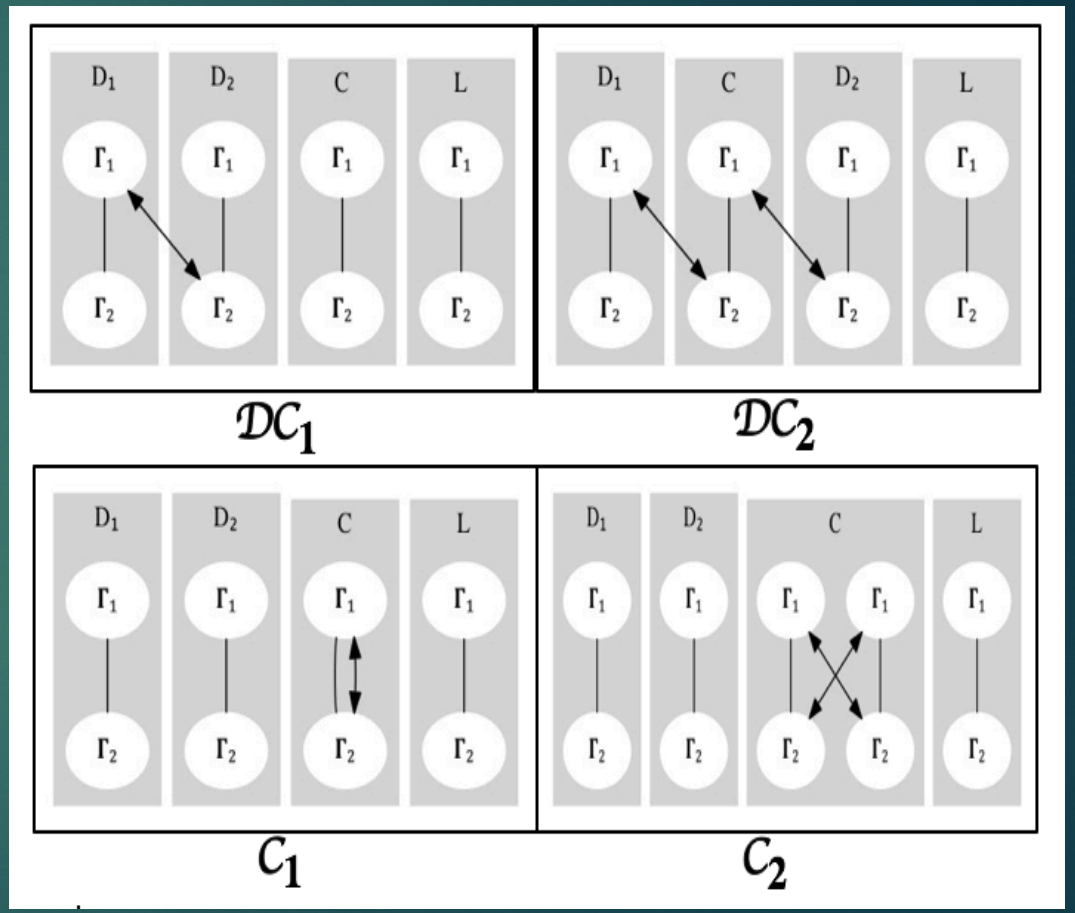
- Partitioning the iteration space ($\mathcal{D}$) in four sets using two references:
  - $\mathcal{D}_1$ iterations share the data used by Γ_α.
  - $\mathcal{D}_2$ iterations share the data used by Γ_β.
  - $\mathcal{C}$ iterations reference data using Γ_α and Γ_β which are referenced in other iterations.
  - $\mathcal{L}$ iterations have no reuse.
- Hence, $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2 \cup \mathcal{C} \cup \mathcal{L}$.



Set Representation of the Classification

Partitioning Technique (contd.)

- After $\mathbf{k^{th}}$ steps of the algorithm:
  - $\mathcal{DC}_{\mathbf{k}}$ **partitions**: $\mathcal{D}_1$ iterations that link to **k-1** C iterations and at the end link to a $\mathcal{D}_2$ iteration.
  - $C_{\mathbf{k}}$ **partitions**: The remaining C iterations that are linked to themselves by $\mathcal{T}^{\mathbf{k}}$.

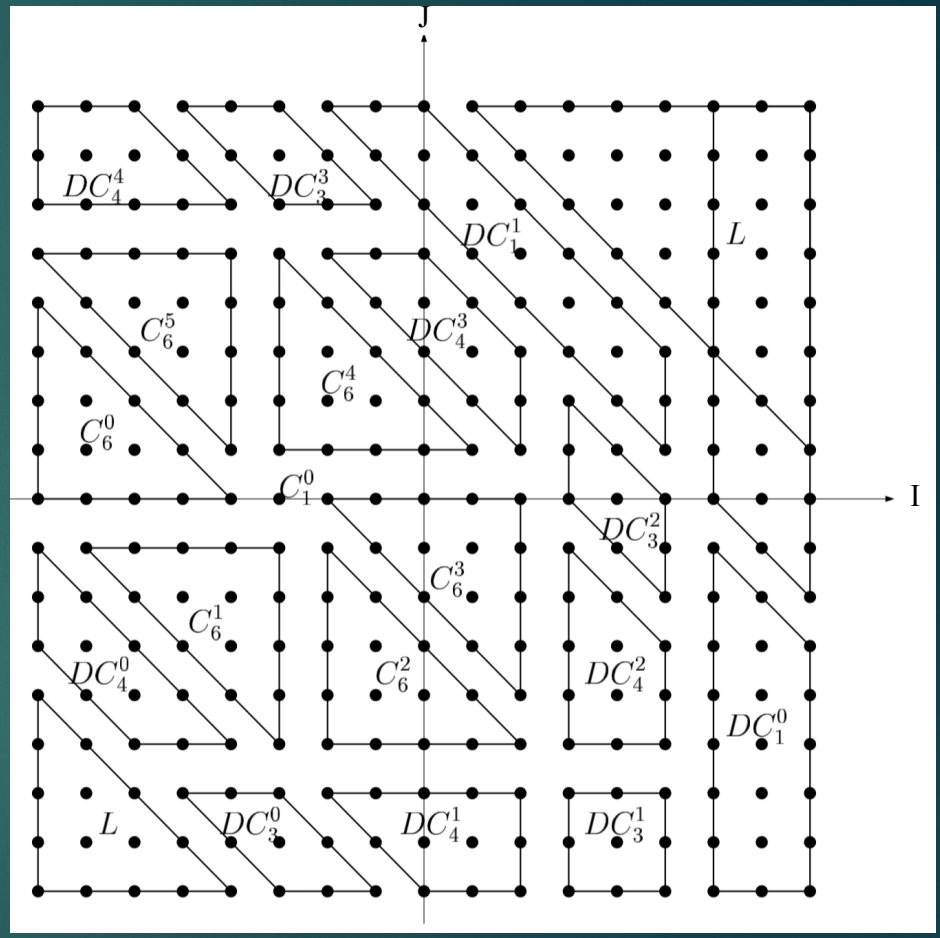


Partitioning Technique (contd.)

- **Halting condition for the algorithm:**

a) If the entire iteration space ($\mathcal{D}$) is completely partitioned. ($\mathcal{T}^{\mathbf{k}} = \boldsymbol{I}$)

b) If $\mathbf{k_{max}}$ is too high then find an *optimal value* of **k** to protect gained speedup.



Adapted from Meister, B. et. al.: The Polytope Model for Optimizing Cache Locality, Technical Report RR 00-03, ICPS-LSIIT (2000)

Code Generation

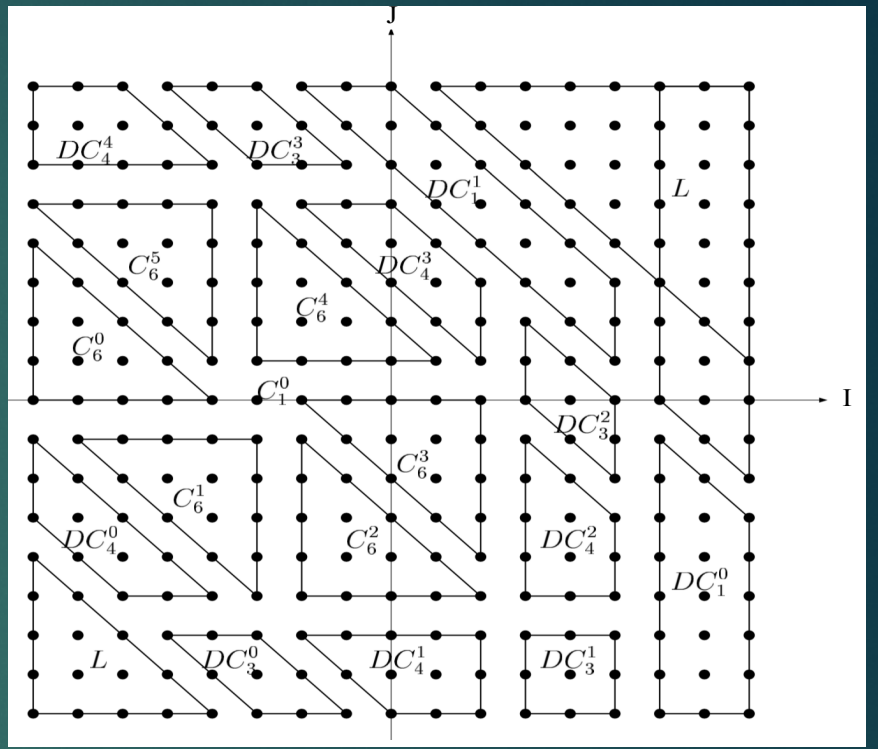
- **First strategy:**

a) Scan the first partition of each type.

b) Generate subscripts for other partitions of similar type using reuse relation: I, $\mathcal{T}$(I), $\mathcal{T}^2$(I), etc.

```
for (i =-N; i <= -4; i++) {
    for (j = MAX(-N+3,-i-N-3); j <= -i-N-1; j++) {
        X[i][j]             = Y[i][i+j+3]     + Y[i+j][j];
        X[-j-3][i+j+3]      = Y[-j-3][i+3]    + Y[i][i+j+3];
        X[-i-j-6][i+3]      = Y[-i-j-6][-j]   + Y[-j-3][i+3];
        X[-i-6][-j]         = Y[-i-6][-i-j-3] + Y[-i-j-6][-j];
        X[ j -3][-i-j -3] = Y[ j -3][-i -3] + Y[-i -6][-i-j -3];
    }
}
```

Index calculation for $\mathcal{DC}_4$ using reuse relation ($\mathcal{T}$).

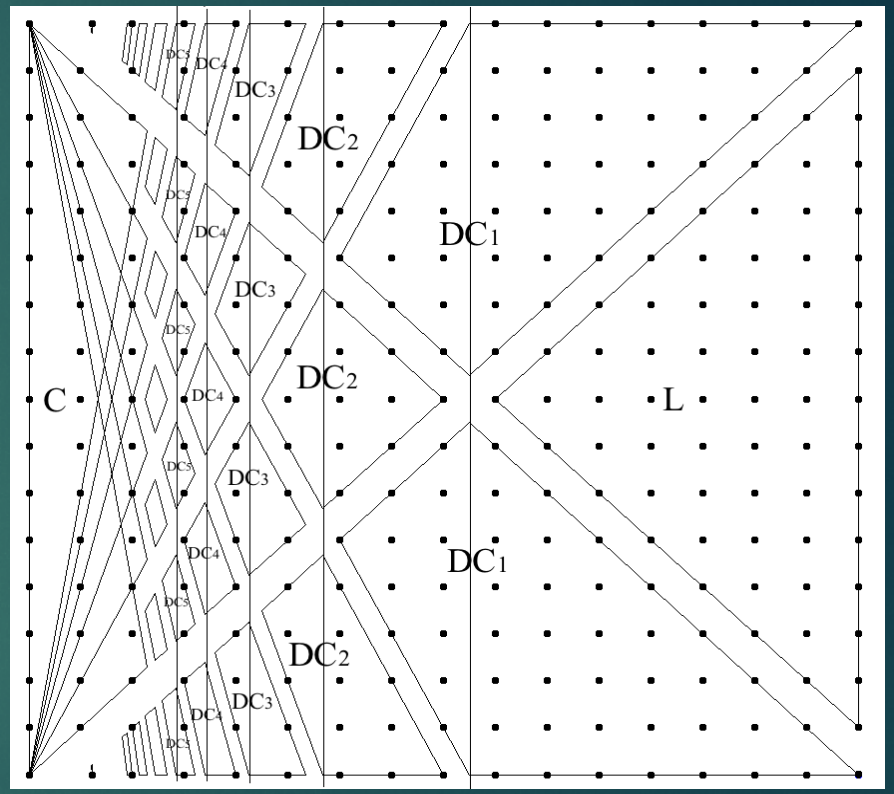


Scanning $\mathcal{DC}_4^0$ to compute index for $\mathcal{DC}_4^1$, $\mathcal{DC}_4^2$, $\mathcal{DC}_4^3$, $\mathcal{DC}_4^4$

Code Generation

- **Second strategy:**

Reduce high control statement overhead by re-partitioning the partitions to reduce boundary check overheads.

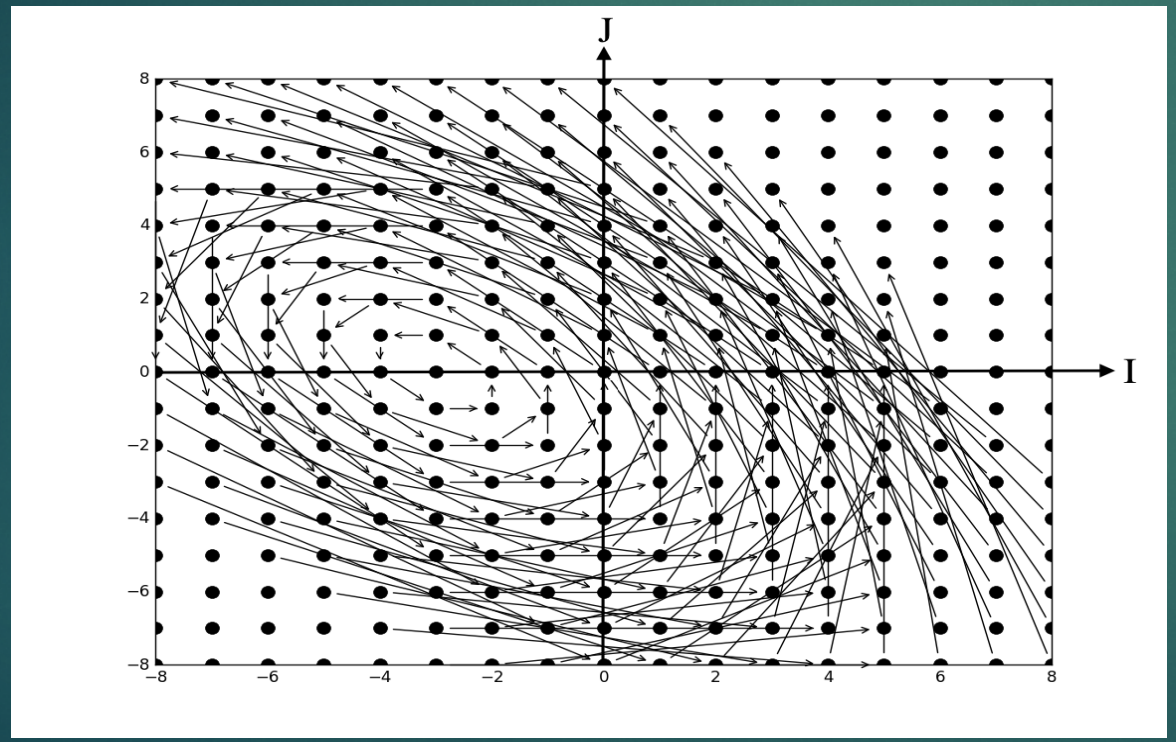


Wave-front Execution of the Polygonal Partitions

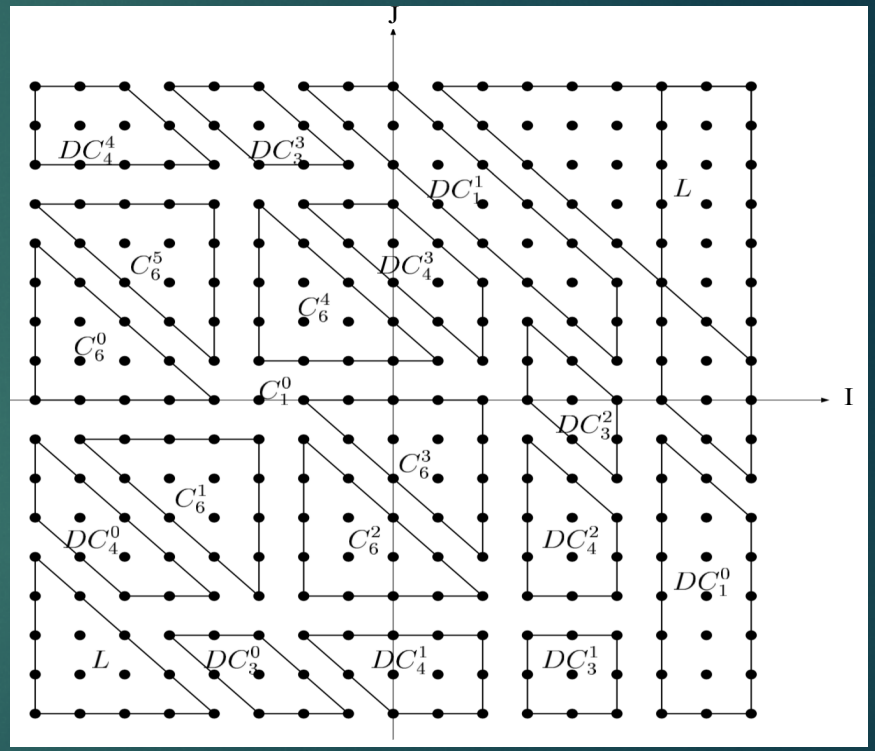
Case 1: Two Dimensional Non-Uniform Reuse Pattern


```
for ( i = -N; i <= N; i++)
        for ( j = -N; j <= N; j++)
                X[i,j] = Y[i,i+j+3] + Y[i+j,j];
```

Loop-Nest



Reuse Pattern



Polygonal Partitions for Two Dimensional Non-Uniform Reuse Pattern ($k_{max} = 6$)

Irregular Scaling of Partitions

Size	$\lvert\mathcal{DC}_4\rvert$	$\lvert C_6\rvert$	Ratio ($\lvert C_6\rvert/\lvert\mathcal{DC}_4\rvert$)
128	1860	47250	**26**
256	3780	192786	**52**
512	7620	778770	**103**
1024	15300	3130386	**205**
2048	30660	12552210	**410**
4096	61380	50270226	**820**

Iteration counts in C6 and DC4 partitions

Partition	Approx. Scaling Factor w.r.t. Dataset
C_6	1x
$\mathcal{DC}_1$, $\mathcal{DC}_4$	0.5x
$\mathcal{DC}_3$, $\mathcal{DC}_1$	0x

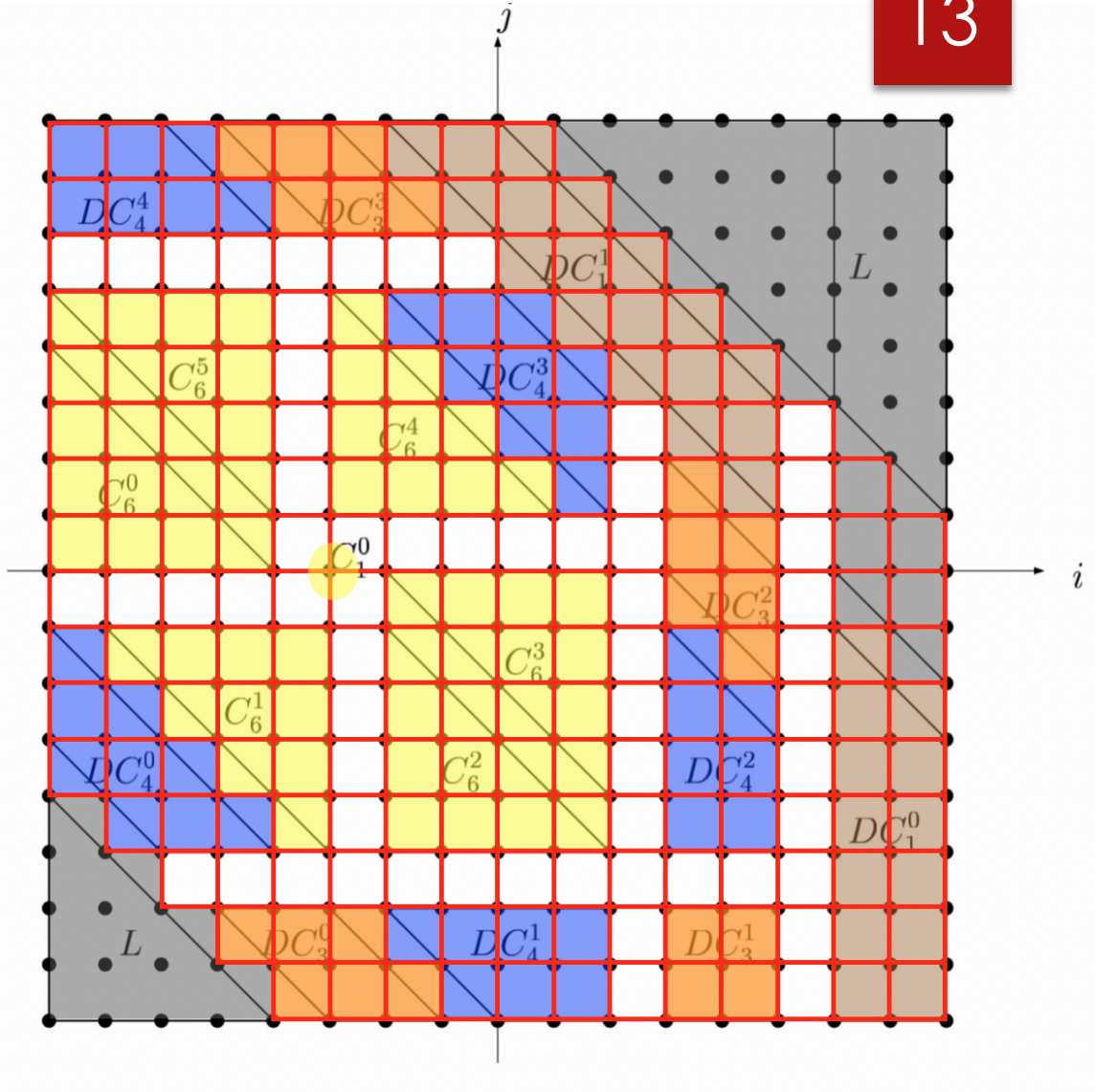
Re-Tiling of Polygonal Partitions

- Load Balancing
  - C_6 partitions execution time dominates the kernel execution time.
- Scalability
  - Scheduling each type of partition on different thread, restricts parallelism.
- Solution for both problems:
  - Re-Tiling the partitions with rectangular tiling.
  - Executing all partitions type one-by-one.
  - Dynamically scheduling re-tiles for a single partition.

Re-Tiling Partitions with Reuse

- $\mathcal{L}$ partitions don't have any reuse.
  - Hence, all iterations can execute in parallel.
- Scheduling partitions based on size.

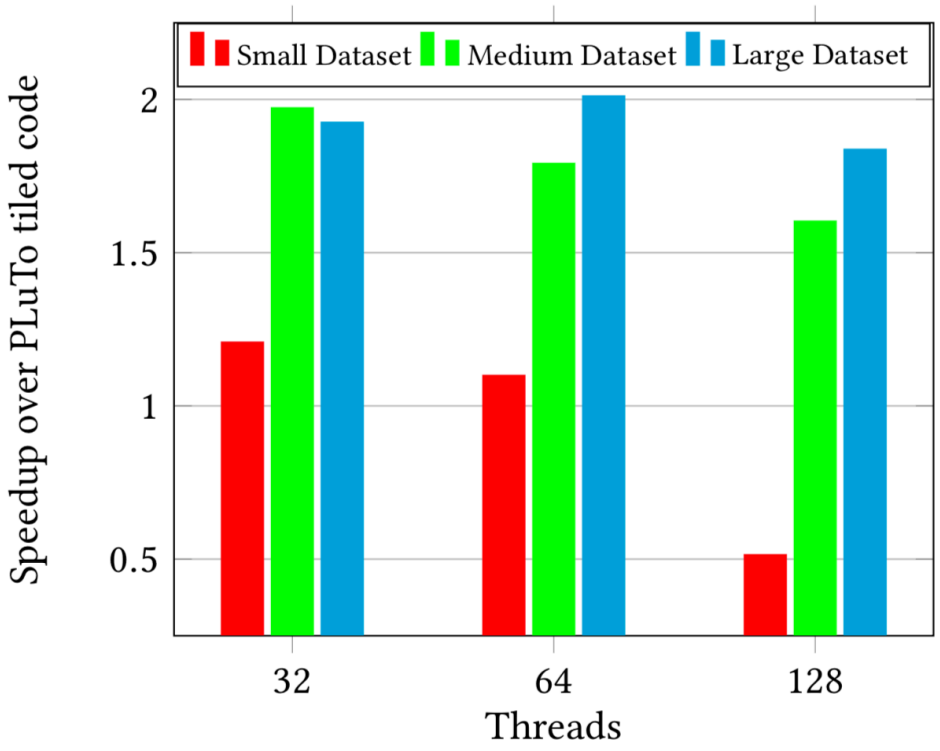
```
#pragma omp parallel for schedule(dynamic)
Loop-Nest : Re-tiled C6 partitions
#pragma omp parallel for schedule(dynamic)
Loop-Nest : Re-tiled DC4 partitions
#pragma omp parallel for schedule(dynamic)
Loop-Nest : Re-tiled DC3 partitions
#pragma omp parallel for schedule(dynamic)
Loop-Nest : Re-tiled DC1 partitions
#pragma omp parallel for schedule(dynamic)
Loop-Nest : Re-tiled C1 partitions
#pragma omp parallel for
Loop-Nest : L partitions
```

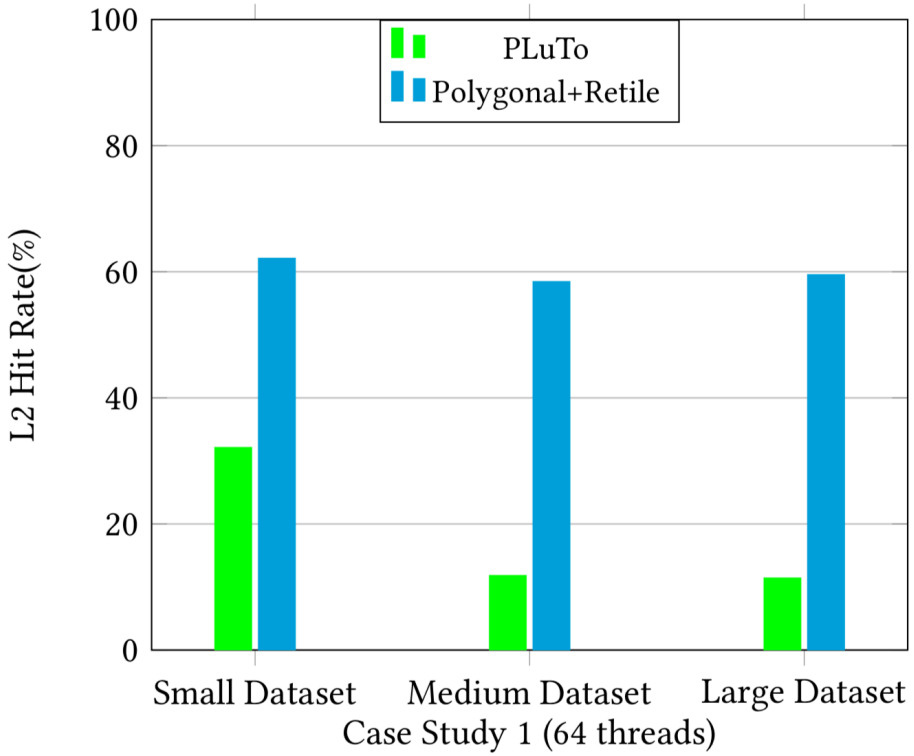


Code Sample after Re-Tiling


```
lbp=ceild(-N-31 ,32);
ubp=-1;
#pragma omp parallel for schedule(dynamic) private(lbv ,ubv,t2 ,t3 ,t4)
for (t1 = lbp; t1 <= ubp; t1++) {
    for (t2 = 0; t2 <= min(floord(N-4,32),-t1-1); t2++) {
        for (t3 = max(-N,32*t1); t3 <= min(32*t1+31,-32*t2-4); t3++) {
            lbv = 32*t2 ;
            ubv = min(32*t2+31,-t3 -4);
            for (t4 = lbv; t4 <= ubv; t4++) {
                x[t3][t4]             = y[t3][t3+t4+3]       +  y[t3+t4][t4];
                x[-t4 -3][t3+t4 +3]   = y[-t4 -3][t3 +3]     +  y[t3][t3+t4+3];
                x[-t3-t4-6][t3+3]     = y[-t3-t4-6][-t4]     +  y[-t4 -3][ t3 +3];
                x[-t3-6][-t4]         = y[-t3 -6][-t3-t4 -3] +  y[-t3-t4-6][-t4];
                x[t4-3][-t3-t4-3]     = y[t4-3][-t3-3]       +  y[-t3 -6][-t3-t4 -3];
                x[t3+t4][-t3-3]       = y[t3+t4][t4]         +  y[t4-3][-t3-3];
            }
        }
    }
}
```

Re-Tiled parallel code for C_6 partition





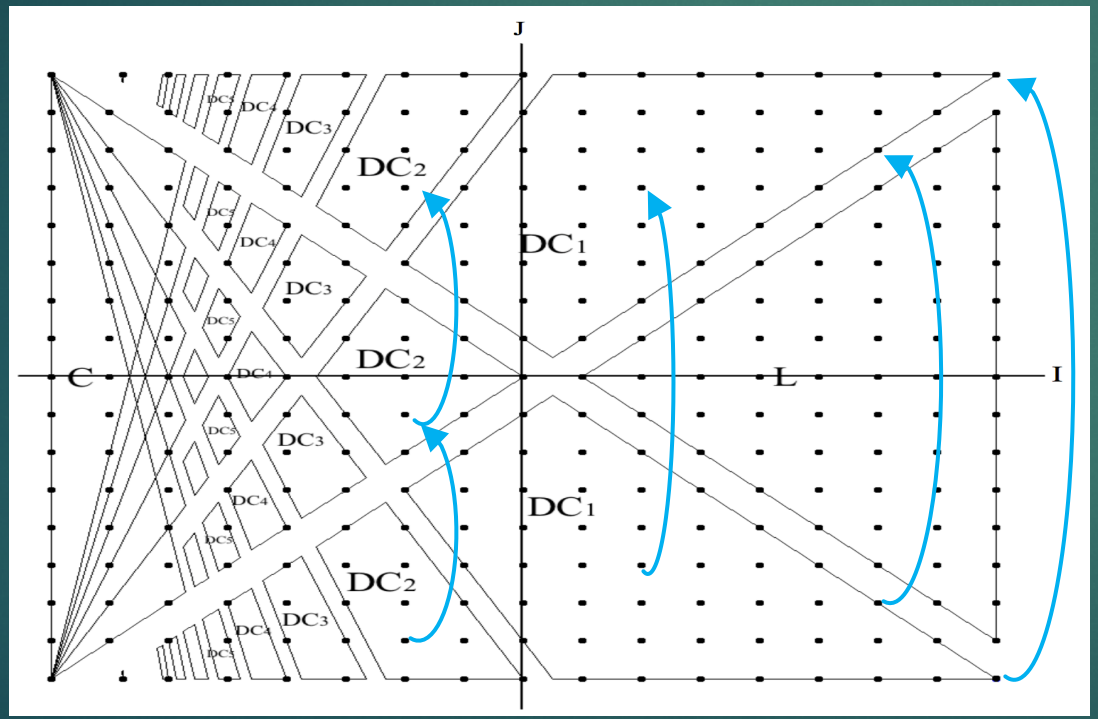
Experimental Results – Case Study 1

Experimental Setup:
Intel Xeon Phi Knights Landing CPU 7210 @ 1.30GHz (64 cores, 1MB L1-cache, 32MB L2-cache) – Quadrant-Cache configuration.
Affinity settings:
OMP_PROC_BIND = spread and OMP_PLACES = threads

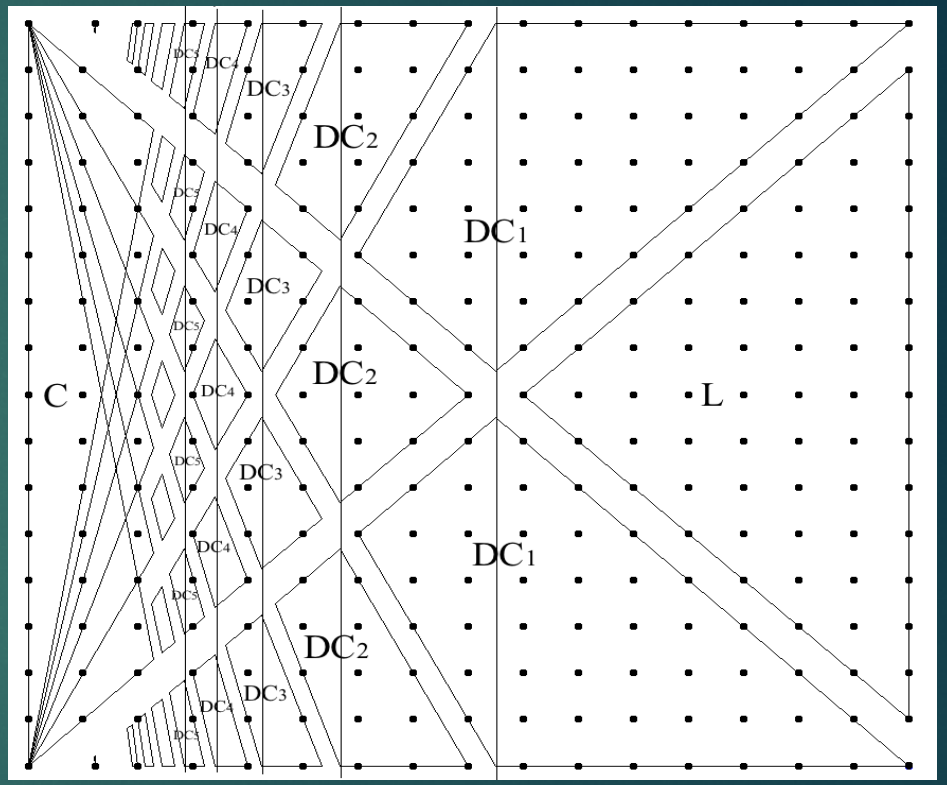
Case 2: One Dimensional Non-Uniform Reuse Pattern


```
for (i = -N; i <= N; i++)
    for (j = -N; j<= N; j++)
        X[i][j] = Y[i][j] + Y[i][i+j+N];
```

Loop-Nest



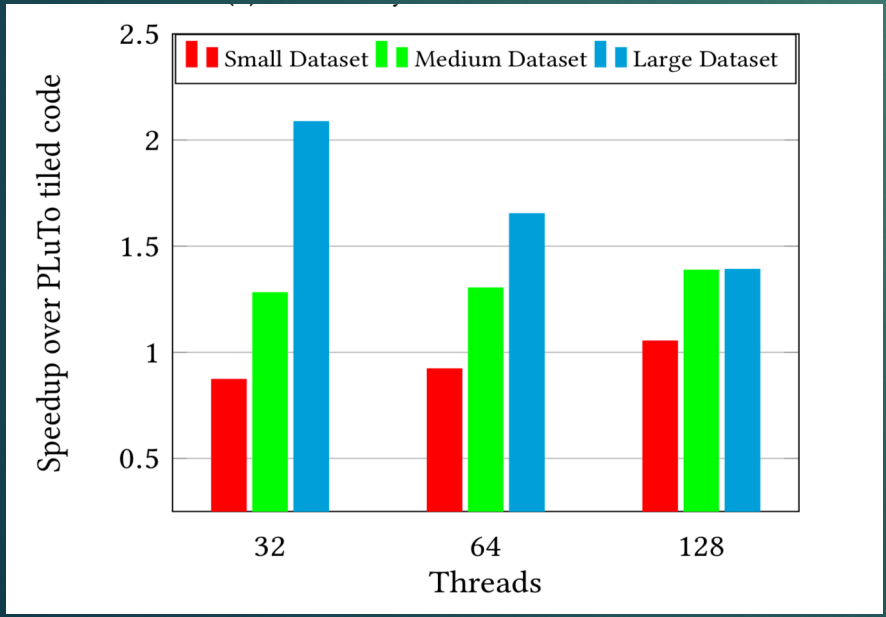
Reuse Pattern

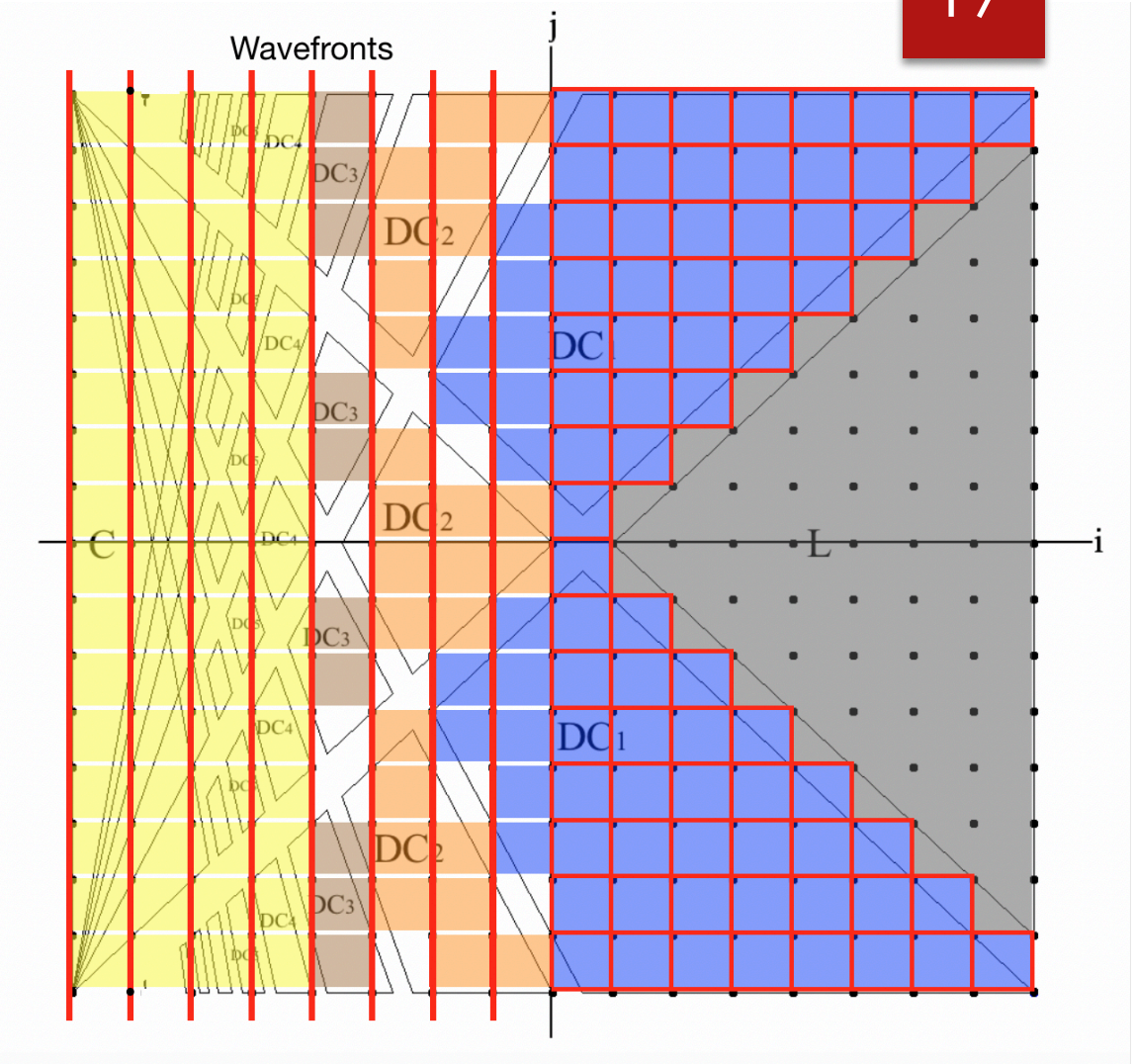


Wavefront Execution for Polygonal Partitions with One Dimensional Non-Uniform Reuse Pattern

Re-Tiling with Wavefront Execution

- Smaller partitions are executed as C type partition.
- Partitions are split to reduce control statement overhead.
- Wavefronts don't hinder reuse.





Summary

- Polygonal tiling technique is **not constrained** to either the **shape** or the **size** of tiles.
- The shapes and sizes are **governed by the reuse pattern** of the loop-nests.
- **Re-Tiling** provides **load-balancing** and **scalability** to the Polygonal Tiles.
- Up to 2x speedup over rectangular tiled code.