Locality, Matrix Multiplication, and Affine Transformations

Advanced Compiler Techniques

Source:
https://www.youtube.com/playlist?list=PLf3ZkSCyj1tf3rPAkOKY5hUzDrDoekAc7
Videos: 125 - 130

Part 1: Locality

Locality: What is it? Why do we want it?

- Data locality refers to data accessed being near to each other, either in the spatial dimension, or the temporal dimension.
- Spatial locality: Addresses which are spatially near each other get accessed. Example: A-1, A, A+1, A+2, etc.
- Temporal locality: Same address is accessed again and again. Example: A, A, B, A, C, A, etc.
- Caches exploit locality to reduce the (average) memory latency observed by the execution pipeline.
- Typically more locality = More hits in caches = Faster Execution!

Interchanging Loops can affect locality

```
for(i = 0; i < 5; i++)
    for(j = 0; j < 10 ; j++)
        x = a[j]
```

Reordering Loops can affect locality

```
for(i = 0; i < 5; i++)
    for(j = 0; j < 10 ; j++)
        x = a[j]
```

High Spatial locality, low Temporal locality

a[6]	a[7]	a[8]	a[9]	a[0]	a[1]

Miss

Reordering Loops can affect locality

```
for(j = 0; j < 10; j++)
    for(i = 0; i < 5 ; i++)
        x = a[j]
```

High Spatial locality, high Temporal locality

a[0]	a[1]				

Miss
Hit

What parameters affect locality?

- Spatial: Prefetchers + CL size
- Temporal: Replacement Policy

Part 2: Matrix Multiplication

Matrix Matrix Multiplication

```
for(row = 0; row < A_ROW_MAX; row++)
    for(col = 0; col < B_COL_MAX ; col++)
        for(idx = 0; idx < A_COL_MAX; idx++)
            C[row, col] += A[row, idx] + B[idx, col]
```

`C[row, col]`: Can be reg allocated. One time cost.
`A[row, idx]`: No point in reg allocating. Might need to pay for every access.
`B[row, idx]`: No point in reg allocating. Might need to pay for every access.

Matrix Matrix Multiplication

```
for(row = 0; row < A_ROW_MAX; row++)
    for(col = 0; col < B_COL_MAX ; col++)
        for(idx = 0; idx < A_COL_MAX; idx++)
            C[row, col] += A[row, idx] + B[idx, col]
```

Worst case: A -> Column major, B -> Row major
Best case: A -> Row major, B -> Column major

Realistically, if both A and B are row major, and `B_COL_MAX` is sufficiently large, every access to `B[idx, col]` misses in cache. $O(n^3)$ misses.

Matrix Matrix Multiplication

```
for(row = 0; row < A_ROW_MAX; row++)
    for(col = 0; col < B_COL_MAX ; col++)
        for(idx = 0; idx < A_COL_MAX; idx++)
            C[row, col] += A[row, idx] + B[idx, col]
```

However, if

1. There are more cache lines than there are rows in B
2. More than 1 element of B can completely in a cache line

The inner two loops will only have to face $O(n^2/C)$ misses when the first column would be accessed where C = Cache Line Size / Size of one element of B = number of columns of B that can fit in the cache.

In best case when we have sufficiently high number of cache lines, all columns of B will fit, reducing the total penalty to $O(n^2/C)$ for all three loops

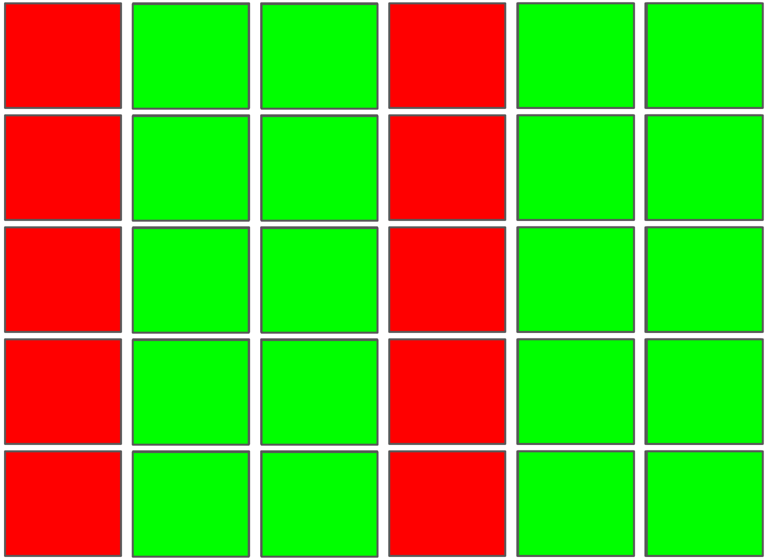
What about A and C?

```
for(row = 0; row < A_ROW_MAX; row++)
    for(col = 0; col < B_COL_MAX ; col++)
        for(idx = 0; idx < A_COL_MAX; idx++)
            C[row, col] += A[row, idx] + B[idx, col]
```

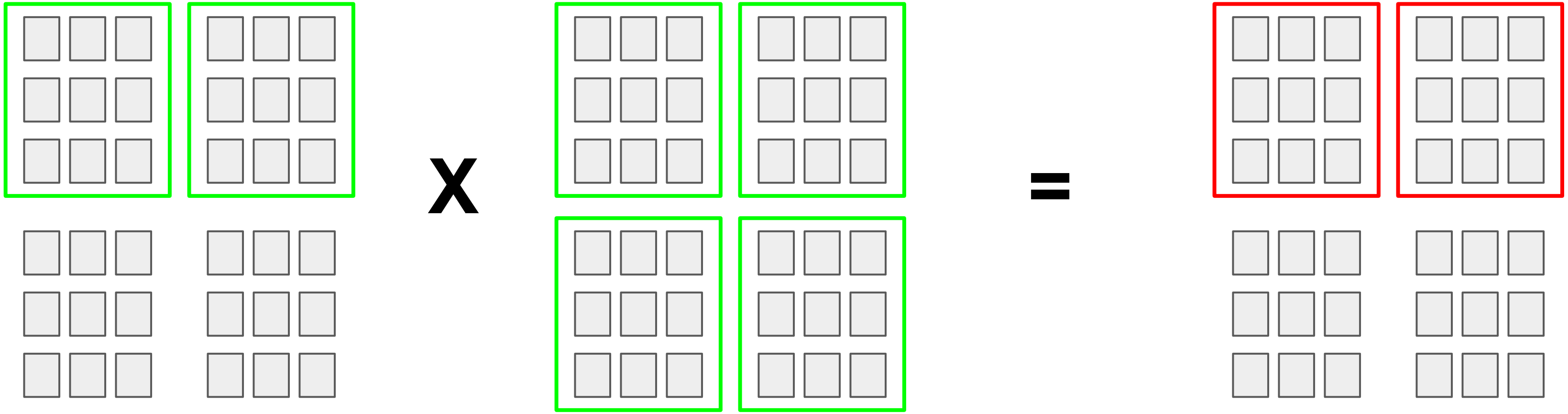
Fastest moving index for both A and C is the columns. Therefore, if A and C are row-major, then we only need to pay $O(n^2/C)$ for each of them.
Therefore in total we pay around $3n^2/C$.

If C > n, we don't get any benefit as then useless data gets fetched into cache. If C = n, we pay around 3n penalty. This makes sense as we need n access for each matrix to bring the matrix into the cache. n accesses because each access brings a complete row, and there are n rows.

Matrix Matrix Multiplication



Blocking Matrix Multiplication



Blocking Matrix Multiplication

Best case, each block faces $3B^2/C$ misses (B^2/C for each sub-matrix). If each element is one byte long:

C = cache line size = l

Total $(n/B)^3$ block operations.

Therefore total cost = $3n^3/BC$

B <= n and C <= n. Therefore, best case we get 3n penalty. This matches the best case estimate from earlier.

Part 3: Affine Loop Transformations

What does affine mean?

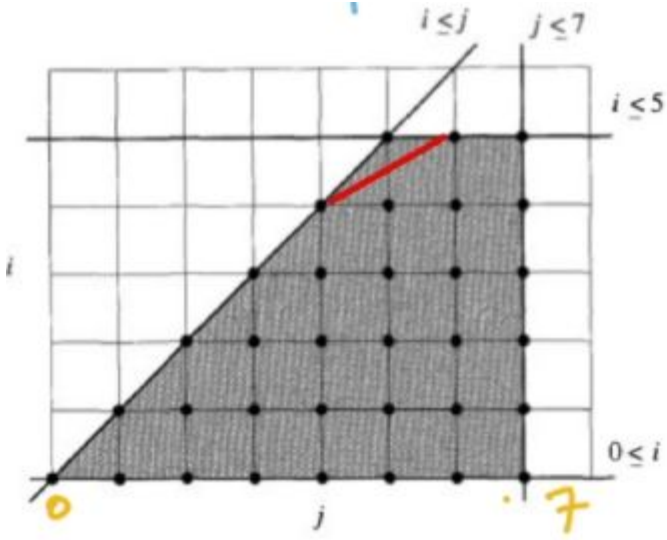
- An affine expression is a linear expression of the inputs.
- $f(x_1, x_2, x_3, \ldots., x_n) = c_0 + c_1x_1 + c_2x_2 + \ldots.. c_nx_n$

When can we use polyhedral optimizations?

- Even if accesses are affine, there might be dependencies.
- Upper and lower bounds of loops are affine functions of outer loop variables.
- Increments are by 1. This can be achieved by using a placeholder variable in loop and multiplying it by a constant before use.
- Under this assumption, the iteration space will always be convex.

Example

```
for (i = 0; i <= 5; i++)
    for (j = i; j <= 7; j++)
        Z[j,i] = 0;
```



Categories of Affine Transformations

- Splitting iteration space into independent slices which can be executed parallely.
- Blocking to create a hierarchy of iterations to improve locality.

Example: Splitting Into Parallel Slices

```
for(idx = 0; idx < N; idx ++)
    a[i] = b[i]
```

Example: Splitting Into Parallel Slices

```
block_size = m
p = ceil(n/m)
for(local_idx = m*p; local_idx < min(m*(p+1), n); local_idx++)
    a[local_idx] = b[local_idx]
```

Affine Transform Theory: Three Spaces

1. Iteration space: Set of all dynamic execution instances, i.e. all possible combinations of iterators. May/May not be rectangular.
2. Data space: Set of array elements accessed. Typically defined as an affine functions of the iteration space.
3. Processor space: Set of all processors in the system. We create an affine function map from iteration space to processor space.

Iteration Space Example

```
for (i = 0; i <= 5; i++)
    for (j = i; j <= 7; j++)
        Z[j,i] = 0;
```

- `i >= 0`
- `i <= 5`
- `j >= i`
- `j <= 7`

Iteration Space Example

```
for (i = 0; i <= 5; i++)
    for (j = i; j <= 7; j++)
        Z[j,i] = 0;
```

- `i >= 0`
- `-i + 5 >= 0`
- `-i + j >= 0`
- `-j + 7 >= 0`

Iteration Space Example

$$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 5 \\ -1 & 1 & 0 \\ 0 & -1 & 7 \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} \geq 0$$

Iteration Space Example

$$\{\underline{i} \in \mathbb{Z}^d \mid B\underline{i} + \underline{b} \geq \underline{0}\}$$

$$\underline{i} = \begin{pmatrix} i \\ j \end{pmatrix} \quad B = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{pmatrix} \quad \underline{b} = \begin{pmatrix} 0 \\ 5 \\ 0 \\ 7 \end{pmatrix}$$

Iteration Space Example: Execution order

- Fastest moving variable is lexicographically smaller.

$$\{\underline{i} \in \mathbb{Z}^d \mid B\underline{i} + \underline{b} \geqslant \underline{0}\}$$

$$\underline{i} = \begin{pmatrix} i \\ j \end{pmatrix} \quad B = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{pmatrix} \quad \underline{b} = \begin{pmatrix} 0 \\ 5 \\ 0 \\ 7 \end{pmatrix}$$

Iteration Space: Loop Invariant

```
for(i=0; i<n; i++)
{
    .....
}
```

$$\{ \underline{i} \in \mathbb{Z} \mid B\underline{i} + \underline{b} \geq 0 \} \ ?$$

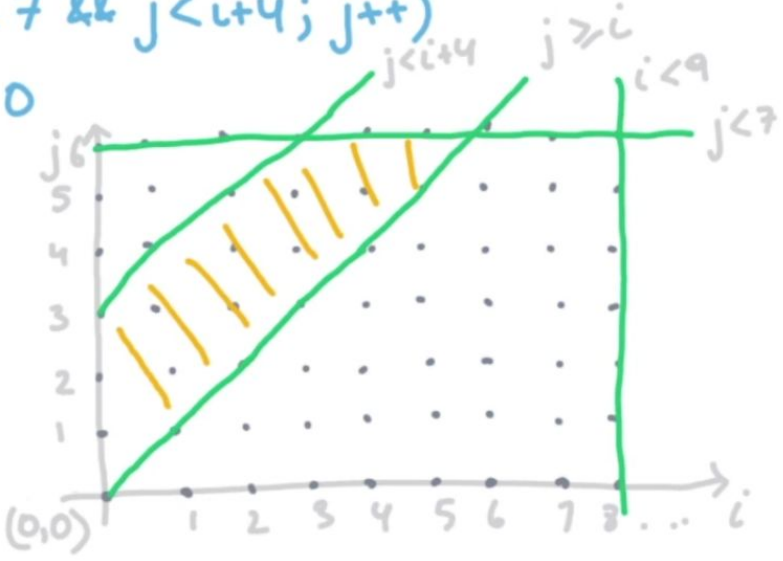
$$\underline{i} = (i) \quad B = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \quad \underline{b} = \begin{pmatrix} 0 \\ n-1 \end{pmatrix}$$

Exchanging Variables

- Suppose we want to exchange i and j

```
for(i=0; i<9; i++)
  for(j=i; j<7 && j<i+4; j++)
    A[i,j] = 0
```

How to switch axes i, j?



Exchanging Variables

1. Project the iteration space on j to find the range of j.
2. For each j, find i in terms of j.

We are guaranteed to have another convex polyhedron after projection.

Exchanging Variables: Projection

- The set of points $(x_1, x_2, \ldots, x_m)$ will be in the projection of S on m dimensions if for some $(x_{m+1}, x_{m+2}, \ldots, x_n)$, $(x_1, x_2, \ldots, x_n)$ lies in S.
- Effective for every point in the projection, you should be able to find some set of values for the rest of n-m dimensions such that the n dimensional point lies in S.

Exchanging variables: Idea

- Project the iteration space onto all the dimensions except the desired innermost variable’s dimension.
- Then project the rest onto all the dimensions except the desired second-innermost variable’s dimension.
- Then project the rest onto all the dimensions except the desired third-innermost variable’s dimension.

… and so on until all variables are exhausted.

Fourier Motzkin Method

Input:

1. A convex polyhedron S in m dimensions.
2. A variable x_m to eliminate

Output:

S’, a projection of S on all dimensions except the m^{th} dimension.

Fourier Motzkin Method: Terminology

$S = \{ \underline{x} \mid B\underline{x} + \underline{b} \geq 0 \}$

C = constraints in S that involve x_m

(coefficient of $x_m \neq 0$)

Fourier Motzkin Method: Algorithm

Algorithm: $S = \{ \underline{x} \mid B\underline{x} + \underline{b} \geq 0 \}$

C = constraints in S involving x_m

For every pair of lower bound and upper bound on x_m in C such that

$$L \leq c_1 * x_m$$

$$c_2 x_m \leq U$$

$$c_1, c_2 \geq 0$$

add

$$c_2 L \leq c_1 U$$

to S'

Also add $S - C$ to S'

Fourier Motzkin Method: Example Setup

```
for(i=0; i<9; i++)
    for (j=i; j< min(7, i+4); j++)
        A[i,j] = 0
```

$$S = \{ \underline{i} \mid B\,\underline{i} + \underline{b} \geq 0 \} \qquad \underline{i} = \begin{pmatrix} i \\ j \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \\ 1 & -1 \end{pmatrix} \qquad \underline{b} = \begin{pmatrix} 0 \\ 8 \\ 0 \\ 6 \\ 3 \end{pmatrix}$$

Eliminate i
(project on j)

Fourier Motzkin Method: Example Solution

$$C = \{ \underline{i} \mid B_c \underline{i} + b_c \geq 0 \}$$

$$B_c = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 1 & -1 \end{pmatrix} \qquad b_c = \begin{pmatrix} 0 \\ 8 \\ 0 \\ 3 \end{pmatrix}$$

$$\begin{array}{ll} 1. & i \geq 0 \\ 2. & i \geq j-3 \\ 3. & i \leq 8 \\ 4. & i \leq j \end{array} \xrightarrow[\text{pair of } \leq \times \geq]{\text{consider each}} \begin{array}{ll} 0 \leq 8 & (1,3) \\ 0 \leq j & (1,4) \\ j-3 \leq 8 & (2,3) \\ j-3 \leq j & (2,4) \end{array}$$

Example Solution

Finally: $S' = \begin{matrix} j \geq 0 \\ j \leq 6 \end{matrix}$ (from S–C)

Thanks!

-Setu Gupta