Optimizing for Parallelism and Locality

Ayush Raina

April 6, 2025

Basic Concepts

If operations can be executed in parallel, they can also reordered for other goals such as locality. If there is no parallelism then there is no chance to reorder operations to improve data locality.

Data Locality

Temporal Locality occurs when same data is used several times within a short period of time. **Spatial Locality** occurs when different data elements that are located near to each other are used within short period of time.

Example for Spatial Locality: An important example when spatial locality occurs is when all the elements that appear on same cache line are used together. The reason is that as so on as one element from a cache line is needed, all the elements in the same line are brought to the cache and will probably still be there if they are used so on. The effect of this spatial locality is that cache misses are minimized,with a resulting imp ortant speedup of the program.

Iteration Spaces

The iteration space of a loop nest is defined to be all combinations of loop index values in the nest.

Constructing Iteration Spaces Each loop has a single loop index which we assume is incremented by 1 at each iteration. If loop increments by $c > 1$ then we can replace uses of index i with $ci + a$ for some integer a and then i is incremented by 1.

```
for(int i = 2;i <= 100; i += 3) {
    Z[i] = 0
}
```

In the above snippet we can see, we are incrementing by 3 everytime. This code sets $Z[2], Z[5], Z[8], \dots, Z[98]$ to 0. We can do the same using following code:

```
for(int j = 0; j <= 32; j++) {
    Z[3j+2] = 0
}
```

where we substituted i with $3j + 2$ with $0 \le j \le 32$.

A d-loop nest can be modelled by d-dimensional iteration space. The dimensions are ordered, with kth dimension representing the kth nested loop counting from outermost loop to innermost loop. A point $(x_1, x_2, ..., x_d)$ in this space represents values for all loop indexes. Consider the following example:

```
for(int i = 0;i <= 5; i++) {
    for(int j = i;j <= 7; j++) {
        /* some work */
    }
}
```

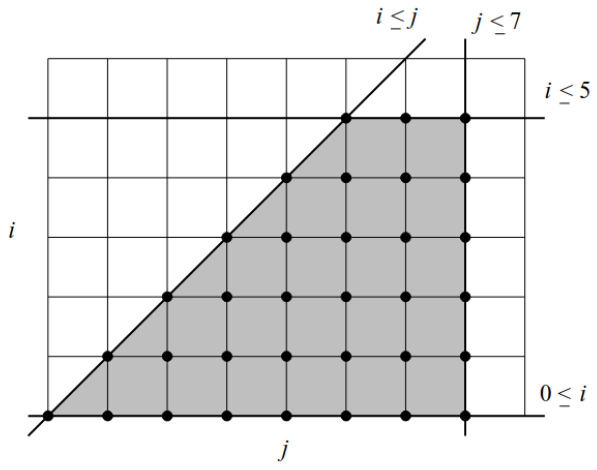

Figure 1: Iteration Space for Nested Loops

This figure represents the iteration space for the code block above.

Execution Order of Loop Nests

A sequential execution of a loop nest sweeps through iterations in its iteration space in an ascending lexicographic order. A vector **i** = $(i_0, i_1, \ldots, i_n)$ is lexicographically smaller then **i′** = $(i'_0, i'_1, \ldots, i'_{n'})$, written **i** < **i′**, iff $\exists\, m < min(n, n')$ such that $(i_0, i_1, \ldots, i_m) = (i'_0, i'_1, \ldots, i'_m)$ and $i_{m+1} < i'_{m+1}$.

$$\begin{array}{cccccccc}
[0,0], & [0,1], & [0,2], & [0,3], & [0,4], & [0,5], & [0,6], & [0,7] \\
 & [1,1], & [1,2], & [1,3], & [1,4], & [1,5], & [1,6], & [1,7] \\
 & & [2,2], & [2,3], & [2,4], & [2,5], & [2,6], & [2,7] \\
 & & & [3,3], & [3,4], & [3,5], & [3,6], & [3,7] \\
 & & & & [4,4], & [4,5], & [4,6], & [4,7] \\
 & & & & & [5,5], & [5,6], & [5,7]
\end{array}$$

Figure 2: Lexicographic Order of Iteration Space

This figure shows the execution order of the loop nest whose iteration space is shown above.

Matrix Formulation of Inequalities

The iterations in d-deep loop can be represented mathematically as $\{i \in Z^d | Bi + b \geq 0\}$. Z is the set of integers, $B \in R^{d \times d}$ integer matrix, b is a integer vector of length d.

Consider the same example as above, we know that $i \geq 0$ and $i \leq 5$. Similarly for j we know that $j \geq i$ and $j \leq 7$. We need to puch each of these inequalities in the form of $ui + vj + w \geq 0$, then $[u, v]$ becomes the row of the matrix B and w becomes the component of the vector b. We can transform these as:

- $i \geq 0 \implies u = 1, v = 0, w = 0$
- $i \leq 5 \implies u = -1, v = 0, w = 5$
- $j \geq i \implies u = -1, v = 1, w = 0$
- $j \leq 7 \implies u = 0, v = -1, w = 7$

Using this, we can formulate the inequalities as:

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

Incorporating Symbolic Constants

Sometimes there are some variables which are loop invariant for all loops in the nest. These are symbolic constants, but to describe the boundaries of an iteration space, we need to treat them variables and **create a new entry for them in vector of loop indexes**.

Consider the following code:

```
for(int i = 0;i < n; i++) {
    // do some work
}
```

In this case n is a symbolic constant, hence our new vector of loop indexes will be $[i, n]$ and iteration space is defined as:

$$\left\{ i \in \mathbb{Z} \;\middle|\; \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ n \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\}.$$

Controlling the Order of Execution

Iterations in the iteration space can be executed in any order as long as their data dependences are satisfied. How to choose an ordering which does not violates data dependences and also optimizes for data locality, we will see in later section of this chapter.

Let us again consider the example as above:

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

In this code, there are no dependences between the iterations. We can therefore execute iterations in the arbitary order in this case sequentially or in parallel. We can see that iteration $[i, j]$ accesses $Z[j, i]$, hence original execution order is like this:

$$\begin{array}{cccccccc}
Z[0,0], & Z[1,0], & Z[2,0], & Z[3,0], & Z[4,0], & Z[5,0], & Z[6,0], & Z[7,0] \\
 & Z[1,1], & Z[2,1], & Z[3,1], & Z[4,1], & Z[5,1], & Z[6,1], & Z[1,7] \\
 & & Z[2,2], & Z[3,2], & Z[4,2], & Z[5,2], & Z[6,2], & Z[7,2] \\
 & & & Z[3,3], & Z[4,3], & Z[5,3], & Z[6,3], & Z[7,3] \\
 & & & & Z[4,4], & Z[5,4], & Z[6,4], & Z[7,4] \\
 & & & & & Z[5,5], & Z[6,5], & Z[7,5]
\end{array}$$

Figure 3: Original Execution Order

But to improve spatial locality, we would like to access the memory locations in the following order:

$$\begin{array}{llllll}
Z[0,0] & & & & & \\
Z[1,0], & Z[1,1] & & & & \\
Z[2,0], & Z[2,1], & Z[2,2] & & & \\
Z[3,0], & Z[3,1], & Z[3,2], & Z[3,3] & & \\
Z[4,0], & Z[4,1], & Z[4,2], & Z[4,3], & Z[4,4] & \\
Z[5,0], & Z[5,1], & Z[5,2], & Z[5,3], & Z[5,4], & Z[5,5] \\
Z[6,0], & Z[6,1], & Z[6,2], & Z[6,3], & Z[6,4], & Z[6,5] \\
Z[7,0], & Z[7,1], & Z[7,2], & Z[7,3], & Z[7,4], & Z[7,5]
\end{array}$$

Figure 4: Improved Execution Order

To get the above order, we should execute the iterations in the following order:

$$\begin{array}{llllll}
[0,0] & & & & & \\
[0,1], & [1,1] & & & & \\
[0,2], & [1,2], & [2,2] & & & \\
[0,3], & [1,3], & [2,3], & [3,3] & & \\
[0,4], & [1,4], & [2,4], & [3,4], & [4,4] & \\
[0,5], & [1,5], & [2,5], & [3,5], & [4,5], & [5,5] \\
[0,6], & [1,6], & [2,6], & [3,6], & [4,6], & [5,6] \\
[0,7], & [1,7], & [2,7], & [3,7], & [4,7], & [5,7]
\end{array}$$

Figure 5: Iteration Order for Improved Locality

The code that executes the iterations in this order is:

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

Projection

Loop Bound Generation is one the many uses of projection. Projection can be formally defined as follows. Let S be an n dimensional polyhedron. The projection of S onto the first m of its dimensions is the set of points $(x_1, x_2, ..., x_m)$ such that for some $x_{m+1}, ..., x_n$ the point $(x_1, x_2, ..., x_m, x_{m+1}, ..., x_n)$ is in S.

Fourier Motzkin Elimination

INPUT: A polyhedron S with variables $x_1, x_2, ..., x_n$. S is a set of linear constraints involving the variables x_i for $1 \leq i \leq n$. One variable x_m is specifies to be eliminated.

OUTPUT: A polyhedron S' with variables $x_1, x_2, ..., x_{m-1}, x_{m+1}, ..., x_n$ that is projection of S onto dimensions other than x_m.

METHOD: Let C be the set of all constraints involving x_m. Do the following:
1. For every pair of lower and upper bound on x_m in C, such as $L \leq c_1 x_m$ and $c_2 x_m \leq U$, create a new constraint $c_1 L \leq c_2 U$. Let us denote set of all these new constraints as C_1.
2. If c_1, c_2 have a common factor divide both sides by that factor.
3. If new constraint is not satisfied, then there is no solution. This means both S, S' are empty spaces.
4. $S' = S - C + C_1$.

If x_m has u lower bounds and v upper bounds, then eliminating x_m will create upto **uv** new constraints.

Example: Consider the figure of iteration space at beginning of Page 2 in this document. Suppose we want to use **Fourier Motzkin** to eliminate i. Then $C = \{i \geq 0, i \leq 5, i \leq j\}$. We can see that C contains 1 lower bound and 2 upper bounds. Hence 2 constraints will be created and $C_1 = \{0 \leq 5, 0 \leq j\}$. Here first constraint is trivial and can be ignored. Then $S' = S - C + C_1 = \{j \geq 0, j \leq 7\}$.

Loop Bound Generation

We compute loop bounds in order from innermost loop index variables to outermost loop index variables. All the inequalities involving innner most loop index variables are written as variable's lower or upper bounds. We then project away the dimension representing the inner most loop to get a polyhedron with one less dimension.

Changing the Axes

Visiting the iteration space horizontally or vertically are just 2 common ways of visiting the iteration space. There are many other possibilities for that. For example we can visit diagonal by diagonal as shown in following example:

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

Iteration space for this code is shown in figure 1 on Page 2. If we want to visit the iteration space diagonally, we have to follow the following order:

$$\begin{array}{llllll}
[0,0], & [1,1], & [2,2], & [3,3], & [4,4], & [5,5] \\
[0,1], & [1,2], & [2,3], & [3,4], & [4,5], & [5,6] \\
[0,2], & [1,3], & [2,4], & [3,5], & [4,6], & [5,7] \\
[0,3], & [1,4], & [2,5], & [3,6], & [4,7] & \\
[0,4], & [1,5], & [2,6], & [3,7] & & \\
[0,5], & [1,6], & [2,7] & & & \\
[0,6], & [1,7] & & & & \\
[0,7] & & & & &
\end{array}$$

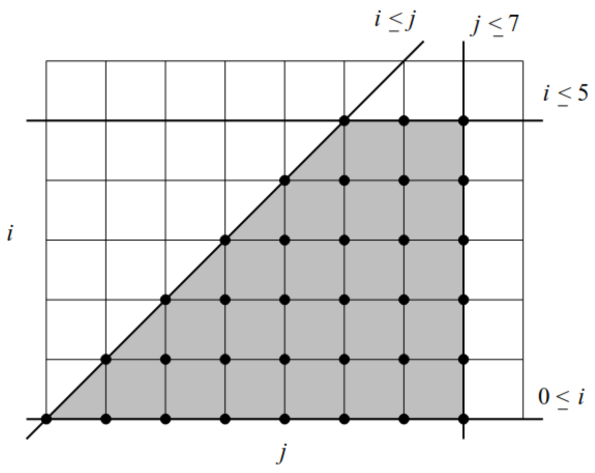

Figure 6: Diagonal Execution Order

We can observe that difference between i, j is a constant in each diagonal starting from 0 to 7. Thus define a new variable $k = j - i$ and we will visit the iteration space in lexico order with respect to k, j now. Hence our contraints become:

1. $0 \leq i \leq 5 \implies 0 \leq j - k \leq 5$
2. $i \leq j \leq 7 \implies j - k \leq j \leq 7$

To create loop bounds we can apply Fourier Motzkin elimination with variable ordering k, j. We get the following constraints:

$L_j = k, U_j = k + 5, 7$ and $L_k = 0, U_k = 7$ as we substituted k accordingly. From these inequalities we get the following code:

```
for(int k = 0;k <= 7; k++) {
    for(int j = k; j <= min(k+5,7); j++) {
        Z[j,j-k] = 0;
    }
}
```

2 Examples to Understand Fourier Motzkin and Loop Bound Generation Completely

Example 1: Reverse the loop nesting order of the following code:

```
for(int i = 1;i < 30; i++) {
    for(int j = i+2; j < 40-i; j++) {
        X[i,j] = 0;
    }
}
```

We can see that loop interchange as a transformation $T(i, j) = (j, i)$. We just need to generate the loop bounds. Consider the following system of inequalities:

$$\begin{aligned}
t_1 &= j \\
t_2 &= i \\
1 \leq i &< 30 \\
i + 2 \leq j &< 40 - i
\end{aligned}$$

We need to first eliminate old loop variables, this can be done in any order. Then start eliminating t_i from the innermost loop to outermost loop to get new loop bounds. In this system on eliminating i we get the following inequalities:

$$t_1 = j$$
$$1 \leq t_2 < 30$$
$$t_2 + 2 \leq j < 40 - t_2$$

On eliminating j we get the following inequalities:

$$1 \leq t_2 < 30$$
$$t_2 + 2 \leq t_1 < 40 - t_2$$

This system can be written as:

$$1 \leq t_2 \leq 29$$
$$t_2 + 2 \leq t_1 \leq 39 - t_2$$

Now we need to eliminate t_2 first to get bounds for t_1. Here the lower bounds of t_2: $\{t_2 \geq 1\}$ and upper bounds for t_2: $\{t_2 \leq 29, t_2 \leq t_1 - 2, t_2 \leq 39 - t_1\}$. Hence newly added constraints are: $\{1 \leq 29, 1 \leq t_1 - 2, 1 \leq 39 - t_1\}$.

Hence we got $t_1 \geq 3$ and $t_1 \leq 38$ as outer loop bounds. Inner loop bounds are $t_2 \geq 1$ and $t_2 \leq min(29, t_1 - 2, 39 - t_1)$. If you carefully observer, we do not need to add $t_2 \leq 29$ as it will be satisfied automatically. Hence our new transformed code is:

```
for(int j = 3; j <= 38; j++) {
    for(int i = 1; i <= min(j-2,39-j); i++) {
        X[i,j] = 0;
    }
}
```

Example 2: Now consider another example with 3 loops:

```
for(int i = 0;i < 100; i++) {
    for(int j = 0;j < 100 + i; j++) {
        for(int k = i+j; k < 100-i-j; k++) {

            X[i,j,k] = 0;

        }
    }
}
```

In this case our transformation is $T(i, j, k) = (k, j, i)$. Here is original set of inequalities:

$$t_1 = k$$
$$t_2 = j$$
$$t_3 = i$$
$$0 \leq i < 100$$
$$0 \leq j < 100 + i$$
$$i + j \leq k < 100 - i - j$$

We can write the same system of inequalities as:

$$t_1 = k$$
$$t_2 = j$$
$$t_3 = i$$
$$0 \leq i \leq 99$$
$$0 \leq j \leq 99 + i$$
$$i + j \leq k \leq 99 - i - j$$

Eliminating i we get:

$$t_1 = k$$
$$t_2 = j$$
$$0 \leq t_3 \leq 99$$
$$0 \leq j \leq 99 + t_3$$
$$t_3 + j \leq k \leq 99 - t_3 - j$$

Eliminating j we get:

$$\begin{aligned} t_1 &= k \\ 0 \leq t_3 &\leq 99 \\ 0 \leq t_2 &\leq 99 + t_3 \\ t_3 + t_2 \leq k &\leq 99 - t_3 - t_2 \end{aligned}$$

Eliminating k we get:

$$\begin{aligned} 0 \leq t_3 &\leq 99 \\ 0 \leq t_2 &\leq 99 + t_3 \\ t_3 + t_2 \leq t_1 &\leq 99 - t_3 - t_2 \end{aligned}$$

Let S be the set of all inequalities then $S = \{t_3 \geq 0, t_3 \leq 99, t_2 \geq 0, t_2 \leq 99 + t_3, t_3 + t_2 \leq t_1, t_1 \leq 99 - t_3 - t_2\}$. We need to eliminate t_3 first.

Constraints involving lower bounds of $t_3 = \{t_3 \geq 0, t_3 \geq t_2 - 99\}$
Constraints involving upper bounds of $t_3 = \{t_3 \leq 99, t_3 \leq t_1 - t_2, t_3 \leq 99 - t_1 - t_2\}$.
Remaining Constraints are: $R = \{t_2 \geq 0\}$.
New Constraints to be added: $C_1 = \{0 \leq 99, 0 \leq t_1 - t_2, 0 \leq 99 - t_1 - t_2, t_2 - 99 \leq 99, t_2 - 99 \leq t_1 - t_2, t_2 - 99 \leq 99 - t_1 - t_2\}$.
Hence new set of constraints $S' = S - C + C_1 = R + C_1$.

Now we need to eliminate t_2 from $S' = \{t_2 \geq 0, t_1 \geq t_2, t_2 \leq 99 - t_1, t_2 \leq 198, 2t_2 \leq 99 + t_1, 2t_2 \leq 198 - t_1\}$.

Again the constraints involving lower bounds of t_2 are: $\{t_2 \geq 0\}$.
Constraints involving upper bounds of t_2 are: $\{t_2 \leq t_1, t_2 \leq 99 - t_1, t_2 \leq 198, 2t_2 \leq 99 + t_1, 2t_2 \leq 198 - t_1\}$.
New constraints to be added are: $C_2 = \{t_1 \geq 0, t_1 \leq 99, t_1 \geq -99, t_1 \leq 198\}$. From this set we can say that $max(0, -99) \leq t_1 \leq min(99, 198) \implies 0 \leq t_1 \leq 99$.

Similarly for t_2 we can say that $t_2 \geq 0$ and $t_2 \leq min(t_1, 99 - t_1, 198, \frac{99+t_1}{2}, \frac{198-t_1}{2})$. A simple analysis shows that $t_2 \leq min(t_1, 99 - t_1)$ is sufficient. Hence we can ignore the other constraints.

And for t_3 we will have $t_3 \geq max(0, t_2 - 99)$ and $t_3 \leq min(99, t_1 - t_2, 99 - t_1 - t_2)$. Here also $t_3 \leq min(t_1 - t_2, 99 - t_1 - t_2)$ is sufficient.

Here is the final code:

```
for(int k = 0;k <= 99; k++) {
    for(int j = 0; j <= min(k,99-k); j++) {
        for(int i = max(0,k-j); i <= min(k-j,99-k-j); i++) {
            X[i,j,k] = 0;
        }
    }
}
```

Affine Accesses

We say that array access in a loop is affine if:

1. The bounds of the loop are expressed as affine expressions of the surrounding loop variables and symbolic constants.

2. The index for each dimension of the array is also an affine expression of surrounding loop variables and symbolic constants.

Example: $Z[i], Z[i+j+1], Z[0], Z[i,i], Z[2i+1, 3j-10]$ are affine accesses. If n is a symbolic constant then $Z[3n, n-j]$ is also affine access. However $Z[i*j]$ and $Z[n*j]$ are not affine accesses.

Each affine array access can be represented by two matrices and two vectors. The first matrix vector pair is **B,b** which we used to represent the iteration space in matrix form.

The second pair is **F,f** which represents functions of loop index variables that produce the array indexes. Formally we represent an array access in a loop nest that uses a vector of index variables $\vec{i} = (i_1, i_2, ..., i_d)$ by the four tuple $\mathcal{F} = (\mathbf{F}, \mathbf{f}, \mathbf{B}, \mathbf{b})$. It maps a $\vec{i} \in \mathcal{I}_s$ to array element location $\mathbf{F}\vec{i} + \mathbf{f}$, where $\mathcal{I}_s$ is the iteration space of the loop nest.

Examples:

1. $X[i-1]$ can be represented as $\begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}i \\ j\end{bmatrix} + [-1]$

2. $X[1, i, 2i + j]$ can be represented as $\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

3. $Y[1, 2]$ can be represented as $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix}$

Data Reuse

Data Reuse: For locality optimization, we wish to identify the sets of iterations that access the same data or same cache line.
Data Dependence: For correctness of parallelism and locality loop transformations, we wish to identify all the dependences in the code. Recall RAW, WAR, WAW dependences.

Consider the following code:

```
1 float Z[n];
2 for(int i = 0;i < n; i++) {
3     for(int j = 0;j < n; j++) {
4         Z[j+1] = (Z[j] + Z[j+1] + Z[j+2])/3;
5     }
6 }
```

Line number 4 can be called as static access where as when we execute different iterations of the loop, then we call it as dynamic access.

Reuse can be classified as **Self** versus **Group**. If the iterations reusing the data come from same static access, then we refer this kind of reuse as **self** reuse. If the iterations reusing the data come from different accesses, then we refer this kind of reuse as **group** reuse.

Further the reuse is **temporal** if same exact location is referenced and it is **spatial** if same cache line is referenced.

In the above example $Z[j], Z[j+1], Z[j+2]$ each have **Self Spatial Reuse** - **self** because they are from same static access and **spatial** because they are very likely to be same cache line. Now Consider iteration j and access $Z[j+1], Z[j+2]$. In the next iteration $j + 1$ we access $Z[j]$ which is same as $Z[j + 1]$ in the previous iteration. Similarly we also access $Z[j + 1]$ which is same as $Z[j + 2]$ in the previous iteration. We can see that we are accessing same data (multiple elements) in different accesses. Hence there is **Group Spatial Reuse** also.

In addition to this, they all have **Self Temporal Reuse** - **self** because they are from same static access and **temporal** because exact same elements are used again and again in each iteration of outer loop. There is **Group Temporal Reuse** also because $Z[j + 1]$ is iteration j is same as $Z[j]$ in iteration $j + 1$. So same memory location is accessed in different accesses.

Degree of Reuse

If the data referenced by a static access has k dimensions and the access is nested in a d-loop nest for some $d > k$ then same data can be reused n^{d-k} times where n is the number of iterations in each loop. For example, if a 3-Loop nest accesses a 2-D array, then the data referenced by the static access can be reused $n^{3-2} = n$ times.

Rank of a Matrix: The rank of a matrix is the number of linearly independent rows or columns in the matrix. For example, consider the matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 4 & 5 & 6 \\ 2 & 1 & 0 \end{bmatrix}$$

In this matrix we can see that $C_3 = 2C_2 - C_1$, Hence there are 2 independent columns here which are C_1 and C_2. Hence the rank of this matrix is 2. We can also see that R_1, R_3 are linearly dependent, from there we can also say that rank of this matrix is 2.

Null Space of a Matrix: A reference in a d-loop nest with rank r accesses $\mathcal{O}(n^r)$ data elements in $\mathcal{O}(n^d)$ iterations, so on average $\mathcal{O}(n^{d-r})$ data elements must refer to same data element. Suppose accesses in this loop nest are represented by **F,f** and **i,i′** be two iterations that access the same data element. Then $\mathbf{Fi} + \mathbf{f} = \mathbf{Fi'} + \mathbf{f}$, which can be written as

$$\mathbf{F}(\mathbf{i} - \mathbf{i'}) = 0$$

If the matrix **F** is fully ranked then $dim(Null(\mathbf{F})) = 0$ and iterations in loop nest access different data elements. Otherwise, $dim(Null(\mathbf{F})) = d - r$. For example consider the same matrix as above:

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 4 & 5 & 6 \\ 2 & 1 & 0 \end{bmatrix}$$

This matrix has rank 2, hence $dim(Null(\mathbf{F})) = 3 - 2 = 1$.

Access	**Affine Expression**	**Rank**	**Nullity**	**Basis of Null Space**
$X[i-1]$	$\begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \end{bmatrix}$	1	1	$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$
$Y[i,j]$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$	2	0	–
$Y[j,j+1]$	$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$	1	1	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$
$Y[1,2]$	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix}$	0	2	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
$Z[1, i, 2*i+j]$	$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$	2	0	–

Table 1: Rank and nullity of affine accesses

In the above table consider the access $Y[i,j], Z[1,i,2*i+j]$, their **F** matrix (also called access matrix) is of rank 2, which means all iterations access different data elements. Hence there is no reuse.

Consider the accesses $X[i-1], Y[j,j+1]$, their **F** matrix is of rank 1. So around $\mathcal{O}(n)$ iterations refer to same location.

Finally consider the access $Y[1,2]$, its **F** matrix is of rank 0, which means all the iterations refer to exactly same location.

Identifying Spatial Reuse

Drop the last row from the access matrix **F** and let us denote this new matrix as **F′**. If the rank of **F′** is $r' < d$-which is depth of loop nest, then we can assure spatial locality by ensuring that innermost loop varies only the last coordinate.

Consider the array access $Z[1, i, 2i + j]$, after deleting the last row, we get the following matrix:

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

This matrix has rank 1, and depth of loop nest is 2.