Compiler Design Notes

Ayush Raina

March 13, 2025

Iteration Space and Iteration Vector

Consider the code block below:

```
for(i = 0;i < N; ++i) {
    for(j = 0;j < N; ++j) {
        /* Do Some Work */
    }
}
```

$I_s = \{(i,j) : 0 \leq i,j < N\}$ and each point (i,j) in this space is called an iteration vector. i,j are called **iteration variables** or **loop induction values**. Iteration vectors tuples contains loop induction values starting from outermost loop to innermost loop.

Consider the following code block:

```
for(i = 0;i < N; ++i) {
    for(j = 0;j < i; ++j) {
        for(k = 0;k < j; ++k) {
            /* Do Some Work */
        }
    }
}
```

In this case $I_s = \{(i,j,k) : 0 \leq i < N, 0 \leq j < i, 0 \leq k < j\}$, and each point (i,j,k) in this space is called an iteration vector. $i,j,k \in \mathcal{Z}$ are called **iteration variables** or **loop induction values**.

Lexicographical Order

$(i,j) < (i',j')$ if $i < i'$ or $i = i'$ and $j < j'$. In General for n dimensional space, $(i_1, i_2, \ldots, i_n) < (i'_1, i'_2, \ldots, i'_n)$ if $\exists\, k \in \{1,2,\ldots,n\}$ such that $i_k < i'_k$ and $i_i = i'_i$ for $i < k$.

Lexicographical Ordering of Iteration Vectors

Consider the following code block:

```
for(i = 0;i < N; ++i) {
    for(j = 0;j < N; ++j) {
        /* Do Some Work */
    }
}
```

The lexicographical ordering of iteration vectors is as follows: $(0,0), (0,1), (0,2), \ldots, (0,N-1), (1,0), (1,1), \ldots, (1,N-1), \ldots, (N-1,0), (N-1,1), \ldots, (N-1,N-1)$. In this course we will plot i on y-axis and j on x-axis.

Lexicographically > 0

Iteration vectors $(a_1, a_2, \ldots, a_n)$ are said to be lexicographically > 0 if first non zero loop induction value is positive. For example, $(0,0,0)$ is lexicographical ≥ 0, $(0,0,1)$ is lexicographical > 0.

Consider the following code block:

```
for(i = 0;i < N; ++i) { /* Loop Header */
    for(j = 0;j < i; ++j) { /* Loop Header */
        A[i][j] = A[i-1][j] + A[i][j-1]; /* Loop Body */
    }
}
```

Above code blocks has N^2 iteration vectors. Hence $N^2!$ possible ways to arrange these vectors. How to find the valid orderings of these vectors?. A valid ordering means an ordering which does not change the semantics of the program.

Dependence Analysis

RAW - Read After Write
WAW - Write After Write
WAR - Write After Read

These are some of the dependencies that can occur in a program. Using this information we can take out the invalid orderings of the iteration vectors. In above program there are 2 reads R_1, R_2, one write W_1. Two instructions are dependent in one of them is write and other is read, same in the case of load and store. In this we have Write(W_1) after Read R_1, R_2. In this program to do W_1 at (i, j) using R_1 at (i', j'), the following conditions must be satisfied:

- $i = i' - 1$
- $j = j'$

Similarly we have dependence between W_1 and R_2. To do W_1 at (i, j) using R_2 at (i', j'), the following conditions must be satisfied:

- $i = i'$
- $j = j' - 1$

Distance Vectors

Consider the following code block:

```
for(i = 0;i < N; ++i) {
    for(j = 0;j < i; ++j) {
        A[j][i] = B[j][i]; /* Copying Array Elements */
    }
}
```

In this case we can do loop interchange to get the following code block:

```
for(j = 0;j < N; ++j) {
    for(i = 0;i < j; ++j) {
        A[j][i] = B[j][i]; /* Copying Array Elements */
    }
}
```

But this will change the execution order, but will this loop interchange will lead to same results ? If we do dependence analysis here we will find that there is no dependence between the two loops because they both are working on different arrays. Hence any ordering of iterations will lead to same results.
Now change B to A in above code to get the following code block:

```
for(i = 0;i < N; ++i) {
    for(j = 0;j < i; ++j) {
        A[j][i] = A[i][j]; /* Copying Array Elements */
    }
}
```

We will test for Read After Write dependence between the two loops. Suppose Read R_1 is at (i, j) and Write W_1 is at (i', j'). To do R_1 at (i, j) using W_1 at (i', j'), the one of the following two conditions must be satisfied:

- $i' \geq i + 1, j' = j, i' = j, j' = i$
- $i' = i, j' \geq j + 1, i' = j, j' = i$

where $0 \leq i, j, i', j' \leq N$. Similarly we can do the same for Write After Read dependence analysis. Suppose Write W_1 is at (j, i) and Read R_1 is at (i', j'). To do W_1 at (j, i) using R_1 at (i', j'), the one of the following two conditions must be satisfied:

- $i' \geq i + 1, j' = j, i' = j, j' = i$
- $i' = i, j' \geq j + 1, i' = j, j' = i$

where $0 \leq i, j, i', j' \leq N$.

Consider the n dimensional space $I_s = \{(i_1, i_2, \ldots, i_n) : 0 \leq i_1, i_2, \ldots, i_n < N\}$. Let $\vec{S}, \vec{T}$ be given. The distance vector between $\vec{s}$ and $\vec{t}$ is defined as $\vec{d} = \vec{t} - \vec{s}$. Then $\forall$ $(\vec{s}, \vec{t})$ st there is a dependence between $\vec{s}$ to $\vec{t}$, then $T(\vec{t}) - T(\vec{s}) > 0$, where T is a linear transformation.

Further if T is a linear transformation then $T(\vec{t}) - T(\vec{s}) = T(\vec{t} - \vec{s}) = T(\vec{d}) > 0$. Hence $\vec{d}$ is lexicographically > 0 if $(\vec{s}, \vec{t})$ is a constant distanc vector.

Fully Permutable Loops

If all the Permutations of distance vectors are constant i.e all the components are positive then the loops are said to be fully permutable. All the Loop Orderings will lead to same results.

Optimizations : Locality

Locality means being close together, we want to focus on data locality which means how close are we to data that we are using. We will talk about Latency and Bandwidth. Latency: Time taken to access a single data element. Bandwidth: Number of data elements that can be accessed in a given time, say per cycle.

Further we have two types of locality:
1. **Locality in Space (Spatial Locality)** : If we are accessing a data element at some particular address, we are likely to access the different address close to it.

2. **Locality in Time (Temporal Locality)**: If we are accessing a data element at some particular address, we are likely to access the same address in near future.

Compiler transformations can enhance locality. Loop Interchange can enhance the Spatial Locality. For example

```
    for(int i = 0;i < 100; i++) {
        for(int j = 0;j < 100; j++) {
            /* Do Some work using f(j,i) */
        }
    }
```

In the above code, even when who cache line is loaded, we are not accessing the data in the order it is stored in the cache line. Hence we can interchange the loops to get the following code block:

```
    for(int j = 0;j < 100; j++) {
        for(int i = 0;i < 100; i++) {
            /* Do Some work using f(j,i) */
        }
    }
```

This will enhance the spatial locality. Following is a example of Temporal Reuse: (Reuse in Time)

```
    for(int i = 0;i < 100; i++) {
        for(int j = 0;j < 100; j++) {
            /* Do Some work using f(i) */
        }
    }
```

In General there are two types of reuse for both spatial and temporal locality - **Self Reuse**, **Group Reuse**

Group Temporal Reuse

Consider the following code block:

```
    load a[i][j];
    load a[i][j+1];
    load a[i][j+2];
```

These kind of access is called Group Temporal Reuse because $a[i][2], a[i][3], ..., a[i][N-3]$ are accessed 3 times. This is called Group Temporal Reuse. These kind of access can also be called as Group Spatial Reuse since we are accessing nearby (**contiguous**) memory locations.

```
    load a[i][j];
```

This of access is called Self Temporal Reuse because $a[i][j]$ is again accessed if put in a loop not depending on i and j.

Matrix Multiplication

Consider the following code block:

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

Assuming all matrices $\in \mathcal{R}^{n\times n}$. Consider Cache Line Size of 64 Bytes, which means it can fit 16 elements of size 4 bytes. Further consider that $N^2 >> C_s$ (Cache Size).

Suppose if there is no Cache, then there are $4N^3$ memory accesses. Now Given Cache, lets try to compute the number of memory accesses which in this case means number of cache misses.

When we Load $A[1][1]$ is accessed then $A[1][1], A[1][2], A[1][3], \ldots, A[1][15]$ are loaded in the cache. For next 15 iterations we will have cache hits. But when we access $A[1][16]$ then $A[1][16], A[1][17], \ldots, A[1][31]$ are loaded in the cache and continue.

Overall we will have $\frac{N^3}{L}$ cache misses for A because of spatial locality. But B does not have spatial locality. Hence we will have N^3 cache misses for B. For N iterations of k loop, $C[i][j]$ is accesses so we assume this is in cache. Hence we will have N^2 loads and N^2 stores for C.

Hence overall memory accesses = $\frac{N^3}{L} + N^3 + 2N^2 \sim \mathcal{O}(N^3)$. Look about how accesses to B can conflict with accesses to A and why we only use 4 way set associative cache.

Suppose now we change the loop order from $(i, j, k) \rightarrow (k, j, i)$, then we will have the following code block:

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

Then N^3 accesses for A, N^2 accesses for B and $2N^3$ accesses for C. Hence overall memory accesses = $3N^3 + N^2 \sim \mathcal{O}(N^3)$. This order is worse than the previous order.

Now Lets consider table with all 6 possible orderings of loops. We will have the following table:

Loop Order	A	B	C
(i, j, k)	N^3/L	N^3	$2N^2$
(k, j, i)	N^3	N^2	$2N^3$
(i, k, j)	N^2	N^3/L	$2N^3/L$
(j, i, k)	N^3/L	N^3	$2N^2$
(j, k, i)	N^3	$2N^2$	$2N^3$
(k, i, j)	$2N^2$	$N^3/16$	$2N^3/16$

This is a example of fully permutable loops. In this case we can interchange the loops in any order and we will get the same results but performance will be different. From above table we can find that (i, k, j) and (k, i, j) are best orderings and they differ in N^2 memory accesses.

March 11, 2025

In the above example of matrix multiplication, with 3 loops. i goes from $1 \rightarrow N$, j goes from $1 \rightarrow N$ and k goes from $1 \rightarrow N$. There are N^3! possible orderings of these loops. We saw 6 of them in the previous example in which dependence is not violated. We want to find out how many are valid execution orders which do not violate the dependencies.

Note: Floating Point operations are not associative. On changing the order, overflow can occur.

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

If k is innermost loop then we have N^2! possible orderings which are all valid. But when Loop if of the form i, k, j then we some more which we need to count.

Given a nested loop and access conditions, we need to find about temporal and spatial locality. Consider the following code block:

```
for(i = 0;i < N; ++i) {
    for(j = 0;j < N; ++j) {
        for(k = 0;k < N; ++k) {
            A[i][j + 2*i + k] /* is accessed */
        }
    }
}
```

We can write the access as matirx(A) vector(V) product where:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 1 \end{bmatrix} \text{ and } B = \begin{bmatrix} i \\ j \\ k \end{bmatrix}$$

Identifying Spatial Reuse

In General for a iteration vector $\vec{i} = (i_1, i_2, \ldots, i_n)$, there is spatial reuse for i_k if k^{th} column has a small non zero element which be possibly negative only in the last row. All other elements in k^{th} column are zero.

Identifying Temporal Reuse

```
for(i = 0;i < N; ++i) {
    for(j = 0;j < N; ++j) {
        for(k = 0;k < N; ++k) {
            A[i][k] /* is accessed */
        }
    }
}
```

In this loop, we have temporal reuse over k loop. (Check and add this)

Consider this code block:

```
for(i = 0;i < N; ++i) {
    for(j = 0;j < N; ++j) {
        A[i+j][2(i+j)] /* is accessed */
    }
}
```

Is there a temporal reuse in this kind of memory access ? In this case around 2N unique elements are accessed but there are N^2 iteration vectors, so there is kind of pseudo temporal reuse (Do they access same element at any point ?)

Consider another code block:

```
for(i = 0;i < N; ++i) {
    for(j = 0;j < N; ++j) {
        for(k = 0;k < N; ++k) {
            A[i][1+j+k][j+k] /* is accessed */
        }
    }
}
```

The access matrix for the above memory access is: $A = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$ and loop vector is $B = \begin{bmatrix} i \\ j \\ k \end{bmatrix}$

Access matrix may not be square matrix. Dimensions of the array can be more than the number of loops. Again if 3 dimensional iteration space containing iteration vectors are accessing a 2D matrix there has to be temporal reuse.

An Access has no temporal reuse if M has full column rank, where M is the access matrix. We can also say like this for $\vec{v} \neq \vec{v}' \implies M\vec{v} \neq M\vec{v}' \implies M(\vec{v} - \vec{v}') \neq 0$.
Hence for temporal reuse $\exists\ \vec{v} \neq \vec{v}'$ such that $M(\vec{v} - \vec{v}') = 0$. This means $(\vec{v} - \vec{v}') \in$ NULL(M).

In the above definitions we were talking about self spatial and self temporal reuse. Consider the following code block:

```
    for(int i = 0;i < N; i++) {
        A[i] + A[i+1] + A[i+2] /* is accessed */
    }
```

This is an example of $\theta(1)$ Temporal Reuse. Most of the Deep Learning Computations have very high temporal reuse $\sim \theta(N^2)$.

Degree of Temporal Reuse

Degree of Temporal Reuse, depends on the rank of the access matrix. If the rank of the access matrix is r then the degree of temporal reuse is $N - r$.

```
for(int i = 0;i < N; i++) {
    B[A[i]]; /* is accessed */
}
```

These kinds of accesses are called indirect accesses. It is not easy to find about locality of these kind of accesses at compile time.

March 13

If we have full column rank for the access matrix, then the transformation is one to one and in this case there will be no temporal reuse.
Today we will look at another loop transformation called Loop Tiling for exploiting Temporal Locality. This is also called Loop Blocking. Let us go back to our old matrix multiplication example.

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

In Loop Interchange (i, k, j) the best we could do is $\theta(N^3/L)$ cache misses. Can we do better ? In the above case we are not getting temporal reuse for B and this will slow us down.

When we need simultaneous reuse along multiple dimensions of the iteration space, here will be using loop tiling. We are given that $c << N^2e$ where e is the number of elements in the cache line and c is cache size. If we carefully observe, to compute row $C[i]$ the $A[i]$ row is fixed and is multiplied with all the columns of B to get the row $C[i]$. Once $C[i][j]$ is computer, now to compute $C[i][j+1]$ the $A[i][0]$ may not be present in the cache due to its small size.

Suppose we do tiling with block size B in horizontal direction then number of cache misses:
$M(A) = \frac{B}{L} \times \frac{N}{B} \times \frac{N}{B} \times N = \frac{N^3}{BL}$
$M(B) = \frac{B^2}{L} \times \frac{N}{B} \times \frac{N}{B} \times N = \frac{N^3}{L}$
$M(C) = \frac{B}{L} \times \frac{N}{B} \times N = \frac{N^2}{L}$, since both read and write is there so $\frac{2N^2}{L}$

Since for B Temporal Reuse is along i direction and I am currently finishing whole $j - k$ plane before moving to next i plane. Now we will take 3D Tile Block.Assumption is if B^2/L elements fits into cache then along i direction, we will have all hits once these elements are loaded. Hence $B^2e <= c$.

$M(B) = \frac{B^2}{L} \times \frac{N}{B} \times \frac{N}{B} \times \frac{N}{B} = \frac{N^3}{BL}$
$M(A) = \frac{B}{L} \times B \times \frac{N}{B} \times \frac{N}{B} \times \frac{N}{B} = \frac{N^3}{BL}$
$M(C) = \frac{B^2}{L} \times \frac{N}{B} \times \frac{N}{B} = \frac{N^2}{L}$, but there is read and write so $\frac{2N^2}{L}$

Hence $e \times (B^2 + B^2 + B^2) \leq c \implies B \leq \sqrt{\frac{c}{3e}}$. Add calculation for B here for different sizes of c, generally $c = 256$kb. From a old paper, the lower bound on cache misses is $\theta(\frac{N^3}{\sqrt{C}})$ for matrix multiplication. In this case Tile Size $T \propto \sqrt{c}$. In this case we choosed same B along all dimensions, but can we choose different B along different dimensions and we need to do the analysis for this.

Point of Diminishing Returns

As we increase the tile size, the increase in cache hit percentage will keep decreasing. Suppose for $B = 2, 4, 8, 16, 32, 64, 128$ then cache hit percentages are $0.5, 0.75, 0.875, 0.9375, 0.96875, 0.984375, 0.9921875$. Hence we can see that the increase in cache hit percentage is decreasing. This is called Point of Diminishing Returns.

Learn about 4 way associative cache and how conflicts will be handled if we use this cache.