Improving Loop Execution using Precise B/F-Ratio Calculation

A thesis submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by

Nitin Sharma

to the
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
May, 2015

CERTIFICATE

It is certified that the work contained in the thesis titled **Improving Loop Execution using Precise B/F-Ratio Calculation**, by **Nitin Sharma**, has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

Amey Karkare 12/5/2015

Prof Amey Karkare
Department of Computer Science & Engineering
IIT Kanpur

Dipankar

Dr. Dipankar Das
Intel Labs
Bangalore

May, 2015

ABSTRACT

Name of student: **Nitin Sharma** Roll no: **13111039**

Degree for which submitted: **Master of Technology**

Department: **Computer Science & Engineering**

Thesis title: **Improving Loop Execution using Precise B/F-Ratio Calculation**

Name of Thesis Supervisor: **Prof Amey Karkare**

Month and year of thesis submission: **May, 2015**

High performance computing requires efficient use of memory. Vast number of application uses array with loops which makes them ideal to study for optimization. We present improvements to an existing cache cost based model to perform precise calculation of distinct cache lines. The count of distinct cache lines help us in guiding loop transformation such as loop interchange. Through this method we get the loop ordering in which the total cache usage is optimal.

We use the parameter B/F-Ratio (Byte – Flop Ratio) in a modified way to set limit on actual system performance. Our approach is different from existing methods in the sense that we consider system parameters like cache size, block size and processor speed. This makes our approach more flexible in adjusting to different systems. We calculate B/F-Ratio of machine, and on the basis of that, we classify our program to be compute or memory bound for that particular machine. Finally, we demonstrate the effectiveness of our approach on programs of *Polybench* benchmark by comparing our results with the ones obtained through simulation of the original programs.

To my parents

Acknowledgements

My thesis advisers Dr Amey Karkare and Dr Dipankar Das's help and guidance ensured that we start this wonderful journey of working with each other. Amey sir has always been ready to give their best effort even in the times when the path on which we were working was not clear. I consider myself blessed to work with him and express my sincere thanks and respect to both of my guides.

I'd like to thank my parents, friends and the others in my life for being patient with me, especially towards the final stretch of thesis completion, when I wasn't at my best socially.

Special thanks to Tejas Gandhi and Shahbaz Khan, IIT Kanpur for taking time to work with me and guide me in a fixed direction. You guys will always be remembered.

Contents

List of Tables

Chapter 1

Introduction

High performance computers requires efficient use of memory. The best memory management may be done with the help of careful hand/manual optimization but this does not solve the problem. A lot of work in automatic transformation has been done. They all have some parameters either some cost based based model or some threshold based on which they guide some loop transformations.

Loop Interchange has been widely studied. It is simply interchanging of loop execution order such that the meaning of the program is not changed, this is checked with the help of dependence vector. We assume our programs to be properly nested. We took an existing cost based DL Model which calculates distinct cache lines used by the program. We propose extension to this DL method.

1.1 Objective

The objective of this thesis is to propose a new parameter called B/F Ratio which can be calculated based on the static/compile time analysis of the program and it can then guide loop interchange. This parameter basically sets an upper bound on system performance. All the programs requiring more performance are classified as compute bound(explained later). For the calculation of B/F Ratio, we have improved the existing DL Model to now exactly calculate the Distinct Access (DA) count along with improvement to the DL count, which is the number of cache lines

used by the program on a particular architecture.

1.2 Motivation for our work

There are significant number of application that contains regular and irregular kernels. Most of them contains loops and array index. If we have perfect loop nesting such that loop interchange is possible then we have to decide which loop ordering would be maximally cache effective. Consider a simple example of matrix multiplication shown in code listing 1.1.

Listing 1.1: matrix multiplication code

```
for(I=0;I<500;I++)
    for(J=0;J<500;J++)
        for(K=0;K<500;K++)
            C[I][J] + = A[I][K] * B[K][J];
```

The following table 1.1 shows the result of keeping each of I,J and K loop as the innermost loop. It can be easily seen from the table 1.1 that for a cache with 512 lines only the choice of I as the innermost loop will fit, since all other loops will cause the cacheto overflow.

Innermost	c	a	b	c+a+b
I	32	32	1	65
J	500	1	500	1001
K	1	500	32	533

Table 1.1: Cache line count for different Innermost loops

1.3 Contribution of Thesis

In this thesis, we proposed and implemented a precise cache cost model which calculates the total cache lines being used by the program. We consider only array accesses since generally they form bottleneck while transferring data to memory. We propose a new parameter called B/F Ratio which can be used to set a machine performance limit. We classify the program into memory bound or compute bound.

If the program is memory bound then loop interchange is performed such that it can reach to minimal B/F Ratio. We then show results on standard benchmark program and compare our cache cost model with simulation results.

1.4 Thesis Organisation

Rest of the thesis is organised as follows:

Chapter 2 discusses the related work with our assumptions for the input program and system model. It basically compares the existing approaches. It gives us an idea of how different models might suit in different situation and what to choose based on your goal.

Chapter 3 presents the background information required for the improvement of existing cache cost model(DL model). We briefly touch Frobenius Problem and use its results to calculate exact distinct access(DA) count. We then present an algorithm for DA count along with algorithm for DL count.

Chapter 4 discusses our new parameter B/F Ratio which sets an upper limit on the system performance. We discuss similar parameters and how B/F Ratio represents a more practical view of existing memory model.

Chapter 5 presents results on standard programs of polybench benchmarks. We discuss our limitation along with our future goals and conclusion of the thesis.

Chapter 2

Related Work

2.1 Program and System Model

Our approach applies to programs in which loop index is affine combination of loop variables, For non affine combination, we take pessimistic approach and assume cache miss for each access. All the array index coefficients are compile time known since we form linear diophantine equation from the array access index. We also have cache line size, cache capacity and other system parameters. For simplicity, we assume only L1 cache to be present and guide transformation according to it. Although this method could be easily extended to handle more than once cache. We define cache overflow as the situation where the program is using more than cache lines than the total capacity of the cache. We assume no reuse of elements in the cache after it has overflown. We assume perfectly nested loops without conditional statements. We do plan to extend this work to handle conditional expression. Execution profiling[1] can be used to determine the probability of execution of each reference, and our estimates can be weighted by such probabilities[2].

2.1.1 Compilation Model

We assume perfectly normalised loop since Loop normalization[3] guarantees unit step size of loops. We are concerned with memory reference by array access only. For

simplicity, we assume arrays are stored in column major format. As for architecture, we assume uniprocessor model with memory hierarchy. We assume least recently used(LRU) policy to be used by cache.

2.2 Related Work

For uniprocessor machine, this problem of counting the total number of cache line was considered by Porterfield in [4]. But the technique assumed cache line size of one element has the total number of distinct element access was the total number of cache line size. No such assumption is made by us. we can solve this special case with exact number of Access. Also they assumed data dependence with constant direction vectors, which is not the usual case in practice [5]. They [4] compute "cache dependence" similar to data dependence. Instead our method use multiple application of the GCD test and properties of linear diophantine equation to better estimate the cache lines.

DL model has been widely used in [6][7][8] for finding the ordering of the loops, loop fusion profitability analysis and determining optimal tile size according to system specific parameters. It can be applied to any level of cache or TLB by selecting its cache line size as page size. In general, to determine the total amount of bytes transferred between main memory and cache we first get a upper bound on the number of cache lines being used by the given program. A lot of work has been done in traditional AST based transformation frameworks for locality and parallelism that builds a memory cost model to calculate cache effectiveness.

AST based transformation are individual loop transformation applied to Abstract Syntax Tree(AST) of the program. In their paper[7] on integrating polyhedral model with AST based transformation. The author make use of DL based analysis model to be used in polyhedral framework and bring advantage of polyhedral model to the world of AST based transformation. A per-iteration memory cost of every statement is calculated which assumes that all distinct lines of a tile are kept on a specific cache. The change in the memory cost with respect to tile size becomes a

driving factor for the selection of loop ordering. The loop interchange is likely to be beneficial when the new cost is smaller than the original per-iteration memory cost. Also for loop fusion the author compares[7] the memory cost before and after loop fusion[6, sarkar].

Another widely known cost based model[9, kennedy] calculates LoopCost Function by partitioning array access into Refgroups. Each of the Refgroup has a representative member array access which is used for all calculation for that particular equivalence class. It classifies access to the same memory location as temporal reuse while access done to consecutive memory location is counted as spatial reuse. But this cost model conservatively assumes that reuse happen only across the innermost loop which greatly decrease its precision. Also their loopcost algorithm appears to be exponential in terms of number of array references in the worst case, in that time our method can give fairly tight upper bound over the number of cache lines that too considering reuse across all present loops. Their work considers overlap of only uniformly generated references, whereas ours is more generally applicable.

The IBM ASTI[6] optimizer was the first of early compilers that performed wide range of transformation using a cost based framework. Before this optimizer High Order(or high level) Transformation used to operate on intermediate representation of the program which was related to the Machine level while this operates on source program level. This was because high order transformation could drastically degrade the performance. It provided foundation for transformation like loop skewing, increase in opportunity for loop-invariant scalar replacement, loop unrolling and loop fusion using a memory cost model.

To guide tile size selection, the paper by P.Sadayappan and Vivek Sarkar [8] presents a new memory cost model since in the existing DL model any tiles with data footprint larger than cache size are discarded, as we conservatively assume that when cache is filled or overflow no reuse will take place which is actually a pessimistic assumption to make in many application. The author[8] introduces the concept of ML(minimum working set lines) that assumes an ideal intra-tile cache

block replacement. Since DL and ML provide lower and upper bound for tile size, it greatly reduces the tile size search space.

Another quantitative analysis has been done by [10, ghosh] where they form cold miss equation(CME's) that give a detailed representation of cache behaviour, focusing more on the cache misses. It actually generates linear diophantine equation similar to our approach but they calculate those reuse which are not found in the cache(hence they calculate all the cache miss) while we on the other hand are interested in finding the total amount of memory traffic that has been brought to the cache from the main memory.

The rest of thesis is organised as follows: In chapter 3, we provide all the background details and terminologies related to DL method along with some new proposed approaches to improve the existing DL method. Chapter 4 then introduces a new concept of byte/flop ratio which is a new approach to control the loop transformation customised to a particular machine. Chapter 5 then extends results of the techniques of previous chapters where we present simulation results for some standard codes. Lastly, we present our conclusion and comment on our future work.

Chapter 3

Background

3.1 Introduction

In this chapter, we provide a basic overview about the linear Diophantine equation and the related concepts to describe their behavior.

3.1.1 linear Diophantine equation

In mathematics, a Diophantine equation is a polynomial equation in two or more unknowns such that only the integer solutions are searched or studied (an integer solution is a solution such that all the unknowns take integer values). A linear Diophantine equation is an equation between two sums of monomials of degree zero or one. For example

$$a_1 * x_1 + a_2 * x_2 + a_3 * x_3, \ldots + a_k * x_k = N \tag{3.1}$$

where N, $a_1, a_2, a_3, \ldots a_k$ are integer constants and $x_1, x_2, x_3, \ldots x_k$ are unknowns that takes integer values.

It is well known that equation 3.1 has integral solution if gcd$(a_1, a_2, a_3, \ldots a_k)$ divides N completely(this is simply extension of Euclid gcd theorem in more than two variables). If we denote by g$(a_1, a_2, a_3, \ldots a_k)$ the largest integer N such that equation 3.1 has no solution in non negative integers, then it is a well-known result

of Sylvester that $g(a_1, a_2) = a_1 * a_2 - a_1 - a_2$. The related functions $n(a_1, a_2, ..., a_k)$ denote the number of positive integers N for which equation 3.1 has no solution[11]. It is also known that $n(a_1, a_2) = \frac{(a_1-1)*(a_2-1)}{2}$. There exist no closed form formula for either g or n [11][12]. More information on this problem may be found in the recently published monograph[13].

3.1.2 Results about Frobenius problem

We will now use two well known Results whose prove is given by respective authors and are helpful in calculating g$(a_1, a_2, a_3....a_k)$ and n$(a_1, a_2, a_3....a_k)$ in general case:

Calculating Frobenius Number

The formula for the calculation of frobenius number has been discussed in [14][11][12]. Let gcd$(a_1, a_2, a_3....a_k) = 1$, and for $1 \leq j \leq k-1$, let m_j denote the least positive integer N congruent to j mod a_1 such that (1) has a solution in non negative integers and all N, $a_1, a_2, a_3....a_k$ are positive integers, then

$$g(a_1, a_2, a_3....a_k) = \max_{1 \leq j \leq a_1 - 1} m_j - a_1 \tag{3.2}$$

$$n(a_1, a_2, a_3....a_k) = \frac{1}{a_1} \sum_{j=1}^{a_1-1} (m_j - j) = \frac{1}{a_1} \sum_{j=1}^{a_1-1} (m_j) - \frac{a_1 - 1}{2} \tag{3.3}$$

Frobenius Number in general

In the previous sub-section we discussed how we can calculate the frobenius number when the $gcd(a_1, a_2 \dots a_k) = 1$ but this might not always be the case, hence in this section we discuss a more general case as given in [15][11][12][16]. the variables $a_1', a_2' \dots a_k'$ denotes the number obtained by taking d common out of the number $a_1, a_2 \dots a_k$. Also note that a_1 is minimum of the coefficients if it is not the case then we can rearrange the terms such that a_1 is minimum.

Let $a_1, a_2, \dots a_k$ be positive integers. If $gcd(a_1, a_2 \dots a_k) = d$ and $a_j = d * a_j'$ for

each $j \leq 1$, then

$$g(a_1, a_2 \dots a_k) = d * g(a'_1, a'_2 \dots a'_k) + a_1(d-1) \tag{3.4}$$

$$n(a_1, a_2 \dots a_k) = dn(a'_1, a'_2 \dots a'_k) + \frac{1}{2} * (a_1 - 1) * (d-1) \tag{3.5}$$

The Problem of Frobenius consist in determining the largest positive integer N that does not have a solution in non negative integers. This is also referred to as "coin change problem of Frobenius". but for our thesis we will be mostly focusing on determining closed form formula for the function n$(a_1, a_2..., a_k)$ which denotes the total number of positive numbers N that does not have a nonnegative solution of equation 3.1. As we can see from equation 3.3 we do not have a closed form formula since equation 3.3 depends on m_j's. If we can find these m_j's then we can put it in equation 3.3 to get the desired results.

3.2 Frobenius Problem with Bounds

It is interesting to see how values are produced by the linear Diophantine polynomial when the input values are bounded i.e they have a lower and upper bound. Consider a very simple code below. The expression in the array index in a linear Diophantine polynomial with $a_1 = 5$ and $a_2 = 7$.

```
for(int i=0;i<=10;i++){
    for(int j=0;j<=10;j++){
        A[5*i+7*j] = A[5*i+7*j]+10;
    }
}
```

Both loop will run 11 times each hence we can expect 11*11 = 121 different values to be produced. But this is not the case,actually only 97 different values are produced and some values are produced twice. If we take all the 97 different values and sort them in ascending order we can notice something interesting that within a certain range every consecutive numbers will be formed. For example in the above

the values produced are

0 5 7 10 12 14 15 17 19 20 21 22 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 98 99 100 101 103 105 106 108 110 113 115 120

It can be seen that between 24 and 96 every consecutive value is produced at least once. We know that since $a_1 = 5$ and $a_2 = 7$ are co-prime then according to Euclid theorem every number should be formed but here some numbers are not being formed since i and j can take only limited values. For example here i and j cannot take negative values. Hence the number of numbers that will be not formed before 24 can be represented by the function $\mathrm{n}(a_1, a_2 \ldots, a_k)$ defined in the beginning of this chapter. Also by symmetry it can be easily seen that same number of numbers cannot be formed in the last. In fact 24 and 96 are at the same distance from the minimum and maximum values respectively. The maximum number that cannot be formed before 24 is 23 which is the Frobenius number in this case and is represented by function $\mathrm{g}(a_1, a_2 \ldots, a_k)$.

Hence we can see that after the point $\mathrm{g}(a_1, a_2 \ldots, a_k)$ every number will be formed till the point $max - g(a_1, a_2 \ldots, a_k)$ and total 2*$\mathrm{n}(a_1, a_2 \ldots, a_k)$ numbers will be missing from $(max - value - min - value)$. Since these missing numbers do not follow any pattern or series there is yet no closed formula for the function $\mathrm{g}(a_1, a_2 \ldots, a_k)$ and $\mathrm{n}(a_1, a_2 \ldots, a_k)$. Also if just one of the coefficient out of $(a_1, a_2 \ldots, a_k$ is 1 then $\mathrm{g}(a_1, a_2 \ldots, a_k) = 0$ and there are no missing numbers. So in that case the total distinct numbers formed by this linear Diophantine polynomial can be given by difference between maximum and minimum value formed.

In the next subsection we will present an algorithm that helps us to evaluate $\mathrm{n}(a_1, a_2 \ldots, a_k)$ and $\mathrm{g}(a_1, a_2 \ldots, a_k)$. As we already know that till date we have no closed form formula for these functions [11]. In fact the general problem is proven to be NP-hard[12].

3.3 Algorithm

The basis of this algorithm is the fact that if we are able to get any number p with the help of left hand side of equation 3.1 then we will be able to get $p+a_1, p+a_2 \ldots p+a_k$ and all other multiples of these constant terms. Without any loss of generality we assume a_1 to be the smallest among $a_1, a_2 \ldots a_k$, otherwise we can rearrange equation 3.1 such that a_1 becomes smallest.

1. create a graph with vertices's $0, 1, 2, \ldots a_1 - 1$.

2. For all k - 1 constants $a_2, a_3, \ldots a_k$ excluding a_1 put an edge from x to $(x + a_n)$ mod a_1 and give them weight $\frac{x+a_n}{a_1}$ where for each value of n, x varies from 0 to $a_1 - 1$ and n varies from 2 to k.

3. Use Dijkstra's Algorithm using 0 as initial node. The distance found out by the algorithm are the numbers that we are searching for.

4. let for any node i the distance found from 0 node is d_i then $d_i * a_1 + i$ is the first number that will be formed with given constants which leaves remainder i when divided by a_i. These numbers have also been defined as $m_j's$ in equation 3.3. We have to find all such $m_j's$ and put their summation in equation 3.3.

5. Thus we have an algorithm which gives us the n($a_1, a_2..., a_k$) which is exactly the count of the values that cannot be produced with non negative values of variables. An important thing to note is that these values cannot be produced due to the constraints of non-negative values of variables otherwise since the gcd$(a_1, a_2..., a_k) = 1$ then every value could have been produced (which is simply Euclid gcd algorithm in two and more than two variables).

3.3.1 Analysis of the Algorithm

In this subsection we will analyse the time complexity of the presented algorithm. Since we apply Dijkstra's Algorithm we will first have to find the maximum possible edges and vertices of our graph.

1. We are taking a_1 to be minimum of all the given coefficients $a_1, a_2, \ldots a_k$ otherwise we can simply rearrange the equation such that the first coefficient become minimum. Thus we can have at most a_1 vertices.

2. We put an edge from x to $(x + a_n)$ mod a_1 where for each value of n, x varies from 0 to $a_1 - 1$ and n varies from 2 to k. hence we can have at most O$(a1 * k)$ edges.

3. The implementation of Dijkstra based on a Min-priority queue implemented by a Fibonacci heap takes running time of $O(|E| + |V| \log |V|)$ (where$|E|$ is the number of edges and $|V|$ is the number vertexes)

4. Hence the time complexity of the algorithm is $O(|a1 * k| + |a_1| \log |a_1|)$ which is theoretically exponential but if the value of a_1 and k is small then for our practical purposes, we will be able to find the exact values of n$(a_1, a_2..., a_k)$.

3.4 Bounding the number of cache blocks

In the previous section we proposed and analysed algorithm related to counting distinct access of a particular array access while in reality we deal in terms of blocks when talking about cache. Hence now we are going to see how to calculate the number of different cache block accessed by a particular array access. Let DA be the number of distinct accesses being made by the array as calculated in the previously presented algorithm. Also Distinct Access(DA) can never be more than product of upper bounds of the loops,hence always take minimum of number of times loop will run and Distinct access obtained. Let MAX be the maximum value of the expression presented as the left hand side of equation 1 if $x_1, x_2, x_3 \ldots x_k$ are the loop variable of k different perfectly nested loops. Also let B be the block size of cache. Since we are dealing with loops that have upper bound and lower bound so there are some numbers which could not be formed due to the constraints of the bounds. Since we have shown in 1.1 that if gcd$(a_1, a_2, a_3....a_k) = 1$ then every number can be formed but these missing numbers are there due to the constraint of the bounds. We denote

the set of these missing numbers by M. It is interesting to see the pattern of these missing numbers which actually helped us to suggest the distinct access algorithm.

Calculating the exact number of cache blocks is more generic problem than calculating the distinct access since the later can be seen as the same problem with unit block size(B=1). Let g be the gcd$(a_1, a_2 \dots a_k)$ and R(range) = MAX-MIN, then we give a close upper bound over the block count(BC) as

$$BC = \min[\frac{DA - 2 * (g(a_1, a_2 \dots a_k) - M)}{B} + 2 * (g(a_1, a_2 \dots a_k) - M), DA, \frac{R}{B}, \frac{R}{g}] \tag{3.6}$$

To extend this equation to multidimensional case we assume as in [17] that accesses can be densely distributed only in the first(least significant) dimension, and are sparsely distributed in higher dimensions. Then number of cache lines can be bounded by the following equation

$$BC(f_1 \dots f_m) \leq BC(f_1) X \prod_{i=2}^{m} BC(f_i) \tag{3.7}$$

If there is possibility of cache line sharing in higher dimension or there is coupling in between two or more dimension then this bound may become an over-estimate which can then solved by applying linearization[18] and using the found expression to get a better bound.

3.5 Exact Block Count

In the previous section we gave an upper bound over the number of distinct cache blocks which is fairly good for small coefficients but the difference becomes significant with large coefficients. In this section we present an algorithm that can give us exact count of the blocks. For simplification let us define a single dimensional access and calculate its Block Count(BC). Consider the following code snippet which will run along with the algorithm for better understanding.

Listing 3.1: code example

```
for(int i=1;i<=10;i++){
    for(int j=1;j<=10;j++){
        A[8*i+11*j] = A[8*i+11*j]+10;
    }
}
```

Let B be the block size and a1 be the smallest of the coefficient of the linear Diophantine polynomial from array index. Let lcm = L.C.M(B,a1). For our example let us take B = 3 and a1 = 8 (from the above code) then lcm = 24. The algorithm is as follows :

1. The first step of the algorithm is to use the previous algorithm 3.3 to calculate the m_j's as defined in section 3.3.

Mod 8	0	1	2	3	4	5	6	7
m_j's	0	33	66	11	44	77	22	55

Table 3.1: Table showing result of step one

2. Form $\frac{lcm}{B}$ groups of size B such that each group contain three rows. First row containing continuous numbers starting from the minimum number formed(banerjee inequality). second row contains numbers equal to first row % a1. The third row contains number from table 3.1 using second row as key values to get the m_j's. The final result of this step is shown in table 3.2.

Number(N)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
N%8	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6
m_j's	0	33	66	11	44	77	22	55	0	33	66	11	44	77	22
table continued															
Number(N)	15	16	17	18	19	20	21	22	23						
N%8	7	0	1	2	3	4	5	6	7						
m_j's	55	0	33	66	11	44	77	22	55						

Table 3.2: Table showing result of step two

3. From each group of size B find the minimum of the m_j's i.e in the third row of table 3.2. Also find the corresponding number(N) from the first row of same

table. For each group if chosen $m_j \leq corresponding(N)$ then contribution by this group will be zero. Otherwise the contribution by this group will be $\frac{m_j - N}{lcm} + test$ where test is zero if $(m_j - N)\%lcm == 0$ otherwise $test = 1$. This value calculated are actually missing blocks and hence summation of this value over all groups will give missing blocks, i.e the number of blocks which will not be referenced at all. So from table 3.3 we get total contribution(C_i) to be 2(if we start from block 0 then block 1 and block 4 will not be accessed).

Min m_j	0	11	0	11	22	0	11	22
Number(N)	0	3	8	11	14	16	19	22
Contribution(C_i)	0	1	0	0	1	0	0	0

Table 3.3: Table showing result of step three

4. Similar number of blocks will be absent from the end. Since as shown in section 3.2 that linear Diophantine polynomial behaves exactly same from both the end. Hence if f^{hi} and f^{lo} are the highest and lowest value of the polynomial then block count is given by

$$BC = \frac{f^{hi} - f^{lo}}{B} - 2 * \sum_{j=1}^{\frac{lcm}{B}} C_j$$

where $\frac{lcm}{B}$ is the number of groups formed of size B(block size) and C_j is contribution of each block towards the missing blocks. There might be difference of 1 or 2 between the actual block count and the one obtained from the above formula since the total elements might not be proper multiple of blocks. But this small difference is easily acceptable in analysis like this. To extend this method to multidimensional array access we can either perform linearization [18] or We can calculate upper bound from section 3.4. The Analysis of the algorithm is fairly easy and gives time complexity of O(a1*B) which is quite acceptable for all practical purposes.

3.6 Bounding Multiple Array References

In the section we will take a look at the effect of one array access on the other accesses. The work in this section is closely related to [2]. Since in reality the blocks brought by one array access will help the other array access if they have to use the same element while they are within our cache and hence haven't been flushed out. We will hence be closely on the array access of the same array since they can only reuse the elements.

consider a very simple code snippet that contains nested loop with a statement with three different array access.

Listing 3.2: code to understand multiple array reference

```
for(int i=1;i<50;i++){
    for(int j=1;j<50;j++){
        A[i+j] = A[2*i+j]+A[i+2*j]+10;
    }
}
```

Adding the number of blocks by each array access is the worst case since in that case we are considering no overlap will take place between any of the array accesses. Another way of looking at the problem is that each of the array index expression is a linear Diophantine expression hence we have to find the total number of blocks that are being used by all these linear expressions over the range of loops. Firstly we consider two one dimensional array references,$f1$ and $f2$. We obtain lower and upper bounds, $f1^{lo} \leq f1 \leq f1^{hi}$ and $f2^{lo} \leq f2 \leq f2^{hi}$ (using banerjee inequality) and the gcds $g1$ and $g2$ of their respective coefficients.

The number of distinct array elements accessed by functions $f1$ and $f2$ is given by $DA(f_1, f2) \leq (\frac{f1^{hi}-f1^{lo}}{g1}+1)+(\frac{f2^{hi}-f2^{lo}}{g2}+1)-OVERLAP(f1, f2)$, where OVERLAP(f1,f2) is the common elements which can be given by $OVERLAP(f1, f2) \leq (\frac{HI(f_1,f_2)-LO(f_1,f_2)}{l}) + 1$, where l is $LCM(g1, g2)$, $HI(f_1, f_2) \leq \min[f1^{hi}, f2^{hi}]$ and $LO(f_1, f_2) \leq \max[f1^{lo}, f2^{lo}]$.

OVERLAP$(f1, f2)$ may be empty, for example in $f1(i) = 2i$ and $f2(i) = 2i + 1$

for $1 \leq i \leq 8$ the OVERLAP$(f1, f2)$ is zero. We can use the gcd test of data dependence[2] to see whether the equation $f1 = f2$ has any integer solution, otherwise we set $TEST(f1, f2) = 0$. Putting it all together we get distinct access as in [2]

$$\begin{aligned} DA(f1, f2) &= DA(f1) + DA(f2) - OVERLAP(f1, f2) \\ &\leq (\frac{f1^{hi} - f1^{lo}}{g1} + 1) + (\frac{f2^{hi} - f2^{lo}}{g2} + 1) \\ &- TEST(f1, f2) \times OVERLAP(f1, f2) \end{aligned} \tag{3.8}$$

We can generalize above equation to more than two references, consider three functions $f1, f2, f3$. Here DA$(f1, f2, f3)$ is given by [2]

$$\begin{aligned} DA(f1, f2) &= DA(f1) + DA(f2) + DA(f3) \\ &- OVERLAP(f1, f2) - OVERLAP(f2, f3) - OVERLAP(f1, f3) \\ &+ OVERLAP(f1, f2, f3) \end{aligned} \tag{3.9}$$

3.7 Exact DA in Multiple Array Reference

Consider the following simple code snippet that contains nested loop with a statement with two different one dimensional array access.

Listing 3.3: code to understand exact DA

```
for(int i=1;i<20;i++){
    for(int j=1;j<20;j++){
        B[5*i+7*j] = B[8*i+11*j]+10;
    }
}
```

In the previous section we showed how we can give a tight upper bound on the number of distinct access with constant time. But if we are ready to give some relaxation in time complexity then we can get the exact Distinct Access. For

simplification let us define our algorithm with two array access in one dimension and then extend it to handle multiple access. Let $f1$ and $f2$ be two array references be $a1 * i_1 + a2 * i_2 \ldots + aK * i_K$ and $b1 * i_1 + b2 * i_2 \ldots + bK * i_K$ such that $a1 = min(a1, a2 \ldots aK)$ and $b1 = min(b1, b2 \ldots bK)$ w.l.g let us assume that $a1 \leq b1$. For example let us take $f1(i, j) = 5 * i + 7 * j$ and $f2(i, j) = 8 * i + 11 * j$ as the two array access where $0 \leq i \leq 20$ and $0 \leq j \leq 20$(just like the code in the start of this section). Following is the step by step algorithm with the running example

1. We need to calculate first numbers(actually formed) whose module with a1 gives $0, 1 \ldots a1 - 1$. these have also been defined as m_j's in section 3.3 and can be easily calculated by Algorithm[3.3]. Table 3.4 shows the value obtained by the algorithm[3.3].

mod 5	Number
0	0
1	21
2	7
3	28
4	14

Table 3.4: Table showing result of step one

2. Similarly for the second polynomial(using same algorithm) we will find the first numbers that are actually formed and give remainder $0, 1 \ldots b1 - 1$ when divided by $b1$, denoted by column name Number (Number(N) in table 3.5). We will form a table of b1 rows and a1 columns such that the first column will contains numbers obtained after doing Modulus by a1 from the column Mod 5. We will also calculate 'step' as b1 mod a1 i.e $step = b1\%a1$ and then fill n^{th} column by adding step(which is $8\%5 = 3$ in this case) to the $(n-1)^{th}$ column and taking modulo $a1$.The resulting table is table 3.5

3. As we have seen in section 3.2 after the frobenius number every number can be formed and the similar pattern is repeated in the end. Here we are trying to find those numbers that are uniquely formed by the second polynomial

Mod 8	Number(N)	(N + k × step)%5				
		k = 0	k = 1	k = 2	k = 3	k = 4
0	0	0	3	1	4	2
1	33	3	1	4	2	0
2	66	1	4	2	0	3
3	11	1	4	2	0	3
4	44	4	2	0	3	1
5	77	2	0	3	1	4
6	22	2	0	3	1	4
7	55	0	3	1	4	2

Table 3.5: Table showing result of step two

N	(N+b1)	(N+2*b1)	(N+3*b1)	(N+4*b1)
0	8	16	24	32
33	41	49	57	65
66	74	82	90	98
11	19	27	35	43
44	52	60	68	76
77	85	93	111	119
22	30	38	46	54
55	63	71	79	87

Table 3.6: Table showing result of step three

but are before the frobenius number of first polynomial. For this, consider N%a1(which is N%5 in this case) as our first column and take the entire row to its right i.e the numbers 0, 3, 1, 4, 2 also take the corresponding Number(N) from the column and b1 repetitively to form a1 numbers including Number(N). In our case the Number(N) is 0 in first row and adding 8(b1) repetitively will give 0,8,16,24,32. Similarly fill the complete table of b1*a1 will give values. The final values are shown in table 3.6.

4. Taking the first rows of Table 3.5 and Table 3.6 we get 0,3,1,4,2 which denotes remainder of N % a1(5) and 0,8,16,24,32 which denotes number obtained by adding b1 to N repetitively. '8' in this example has corresponding entry '3' in table 3.5 and the corresponding entry of '3' as key in table 3.4 will give '28' as the number. This actually tells 28 is the first number formed by the first polynomial which gives 3 remainder when divided by 5 while 8 is the

first number formed by the second polynomial which gives 3 remainder when divided by 5. Hence when both polynomial are merged 8 will be added uniquely by second polynomial. The total number of such number added will given by $(28-8)/a1*b1+Test$ where Test = 0 if $(28-8)\%a1*b1 == 0$ otherwise it is 1. Since $a1*b1 = 8*5 = 40$ in this case and $(28\%40 = 28)! = (8\%40 = 8)$ so Test = 1. So the total contribution by this particular entry is $(28-8)/40+Test = 1$. Similarly Contribution of each entry will be calculated, of course if entry from table 3 is larger i.e if 8 had been larger than 28 that denotes zero contribution by the second polynomial. Table 3.7 shows the completely filled entries.

N	(N+b1)	(N+2*b1)	(N+3*b1)	(N+4*b1)
0	1	1	0	0
0	0	0	0	0
0	0	0	0	0
1	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Table 3.7: Table showing result of step four

Adding all the entries of table 3.7 above gives 3. Hence 3 numbers(which are 8,11,16) are added uniquely by the second polynomial(8*i+11*j) before the frobenius point/number of the first polynomial which is 23(since 23 is frobenius number for 5*i+7*j) in this case. Let us call this contribution as C1.(hence C1 = 3 in this example)

5. Similarly numbers will be added in the end. Since we have upper bound also that we cannot cross. The interesting property of section 3.2 comes handy here. Since the numbers behave the same way in the last except that they are shifted due to the absolute values of upper bound. Hence we find the difference(diff) between their absolute maximum values which is $diff = |(a1 * ub1 + a2 * ub2) - (b1 * ub1 + b2 * ub2)|$ where $|\ |$ denotes taking absolute value and ub1 and ub2 are upper bounds of i and j respectively. diff here is

$8 * 20 + 11 * 20 - 5 * 20 + 7 * 20 = 380 - 240 = 140$ we add this diff to the polynomial having lower maximum value such that they can be aligned. So in this case the newly obtained polynomial are $5 * i + 7 * j + 140$ and $8 * i + 11 * j$

6. We have to repeat this entire algorithm to obtain contribution of the other polynomial to the primary polynomial (which is always the one having lowest first coefficient of all). But running first step i.e calculating m_j's can be tricky since there is a added constant (diff) which was not handled in section 3.3.

7. The slight change in the algorithm of 3.3 is that one more node with label diff will be created which will have an outgoing edge to diff%a1 vertex and weight of diff/a1. Now this diff node is considered as source node which was '0' previously. After this all the step remains same and we can get all the m_j's. If initially lower bound of the polynomials was not same then we would have to shift the smaller one so as to align the lower bound in exactly the same way as we did for the upper bounds. We can then run all the steps of this algorithm to obtain the Contribution at the end. Let us call this contribution as C2.

8. Hence the total number of Distinct Access is Distinct Access by primary polynomial which can be easily calculated by algorithm 3.3 + C1 + C2. If there had been more than two access similar C1, C2 would have been calculated for them. The Time Complexity of this algorithm is clearly O(a1*b1) which is exponential but can work for practical purposes.

To handle the multidimensional case we perform linearization [19] of the array into a single subscript expression, then treating the reference as one-dimensional. The above algorithm will perform good in the cases where the upper bound(UB) are much larger than the coefficients i.e a_i's and b_i's. If that is not the case or the smallest coefficient in the primary polynomial is 1(or very less such as < 5) then the difference between exact and upper bound(presented in the section before this) may not be much. Hence in that case it is advisable to use upper bound since it takes O(1) time and is fairly easy to understand.

3.8 Bound on the Number of Distinct Blocks

The work in this section is closely related to [2]. As one might guess calculating the number of Distinct Blocks using Distinct Access is trickier than for single reference since multiple reference may yield elements which are not same but that fall within the same cache line,even if the functions have no elements in common. An important issue in computing BC_{total} is that of considering cross-reference locality[17] for multiple reference of the same array. For example, consider the functions $f1(i) = 2i$ and $f2(i) = 2i + 1$ for $1 \leq i \leq 100$.

The range of functions values will can be split into three subranges: the first containing only the function with the smaller minimum value; the second containing both functions; and the third containing only the function with the larger maximum value. The three subranges of interest are as follows :

$S_1 : \min[f1^{lo}, f2^{lo}] \leq f1, f2 < \max[f1^{lo}, f2^{lo}]$

All accesses in this subrange must either come from function $f1$ or from function $f2$, depending on which of $f1^{lo}$ or $f2^{lo}$ is smaller. Therefore, the number of lines accessed in this subrange can be bounded by the formula for the single reference case(3.4 or 3.5).

$S_2 : \max[f1^{lo}, f2^{lo}] \leq f1, f2 \leq \min[f1^{hi}, f2^{hi}]$

Let $f^{lo} = \max[f1^{lo}, f2^{lo}]$ and $f^{hi} = \min[f1^{hi}, f2^{hi}]$. Since both functions have accesses in this subrange hence

$$BC(f1, f2) \leq \min[DA(f1, f2), \lceil + \rceil 1]$$

where DA$(f1, f2)$ is bounded as in 3.7 or in 3.6, B is the number of maximum elements that the block can contain.

$S_3 : \min[f1^{hi}, f2^{hi}] < f1, f2 \leq \max[f1^{lo}, f2^{lo}]$

Like S_1, all accesses must come from single access. hence blocks in this case should be easily counted by formula for single reference case(3.5 or 3.4).

An upper bound on the number of blocks can be obtained by summing up the

values obtained in different subranges $BC \leq BC(S_1) + BC(S_2) + BC(S_3)$.As shown in [2] this technique can be generalized with three or more references. However, the worst case the execution time of this estimation can be exponential in the number of references to the same array variable. If we require less time taking method then we can assume 100% overlapping array references that are uniformly generated[20] and zero overlap otherwise. Basically we are partitioning array references and computing BC_{total} by adding block count by only representative array reference from each equivalence class[6].

Chapter 4

BF Ratio

Computer architecture sees an unstoppable increase in parallelism computation power while bandwidth grows much more slowly[21]. This has led to the trend where algorithm designers are adjusting their approach by reducing data movement and synchronization points. This approach by designers is locally optimised and hence might not help in all cases. Also, some cases following this approach might require complete reformation of the existing solution. The pressure from evolving processing speed is so huge that soon algorithms with large communication and synchronization requirement may become obsolete[22].

The slow growth of latency and bandwidth of a particular architecture have forced manual optimization. Let us understand the difference between the bandwidth and latency as explained by [23]. As an analogy if water comes out of end of fire hose 2 seconds after hydrate is turned on at the rate of 1 gallon/sec then the latency of system is 2 seconds while bandwidth is 1 gallon/sec. To put off large fire high bandwidth is required while small fire fighting might require less latency.

4.1 Existing performance metric

Most of the optimization in the past has been focused on general architecture. Optimization customised to a particular platform has been studied in [24] but it still needs attention, since previous approaches did not cover all the transformations. The

goal of such studies is develop technique that automatically restructure programs loops. It statistically estimate the performance of the program and then looks at the objective function to determine whether it is profitable to apply a particular loop transformation.

A method to statically estimate the performance of a given loop on a particular architecture has been presented by [24, Carr]. We analyse a simple transformation of loop interchange and how this method can be used to guide this simple transformation. Although loop interchange has been studied extensively in the past, it has not been shown how to tailor loop interchange to specific loops runs on specific architecture. The paper by Carr[24] present a parameter to measure maximum performance limit of a architecture. They call it 'Machine Balance' while to maintain uniformity with our remaining section, we will refer this as Machine B/F ratio or 'MBF' ratio. This is defined as the steady state in which both memory access and floating point operations are being performed at peak speed[24]. Similarly it defines balance of loops as 'loop balance' measured in bytes/flops, defined as

$$\text{loop balance} = \frac{\text{Number of memory references}}{\text{Number of flops}} \tag{4.1}$$

From now, we call this as Loop B/F ratio or 'LBF' ratio. Based on these performance metric any program can be divided into two categories. If a program on a particular architecture has MBF $<$ LBF then it is referred as Memory Bound, since the loop needs data at higher rate than the machine can provide and idle computational cycle will exist. The performance of such program is to be increased by decreasing the LBF which may be done by decreasing the memory references and/or increasing floating point operations. While if LBF $\leq$ MBF then it is referred as Compute Bound, since data cannot be processed as fast as it is supplied to the processor. Hence performance gain on the same architecture is not possible unless the computing power in increased.

Vendor	Microarchitecture	Model	Byte/s	Flop/s	Byte/flop
Intel	Sandy Bridge	Xeon E5-2690	51.2	243.2	0.211
AMD	Bulldozer	Opteron 6284 SE	51.2	217.6	0.235
AMD	Southern Islands	Radeon HD7970(GHz Ed.)	288	1010	0.285
NVIDIA	Fermi GF110	Tesla M2090	177	665	0.266
IBM	PowerPC	PowerPCA2(BG/Q)	42.6	204.8	0.208
Fujitsu	SPARC64	SPARC64 IXfx (FX10)	85	236.5	0.359

Table 4.1: Table BF ratio of modern microprocessors[26].

4.2 New Performance Metric

Previous Methods[24] use methods like Scalar replacement which can be automatically taken into account by our memory cost model. It uses Total Array access as the measure of memory transfer which actually does not represent the memory view since the data is transferred in terms of cache lines. For example, if the access funtion is $F(i) = 100 * i$ then for $1 \leq i \leq 100$ there are 101 distinct memory access while actually data transferred will be 101*B bytes, where B is the cache line size in bytes. Even the model used for counting distinct array access is not exact while we present better estimation methods along with methods to calculate the total blocks accessed by the kernel. We thus slightly modify their definition of LBF(to take account for cache line transferred) which is somewhat similar to the one defined in Roofline model [25] which is a new performance model that sets an upper bound on performance of kernel depending on the kernels 'operational intensity'. Hence the modified definition is shown in equation 4.2

$$LBF = \frac{(\text{Distinct Cache Lines}_{total}) * (\text{Cache line size})}{\text{Floating point operations}_{total}} \tag{4.2}$$

4.3 BF ratio and Time Complexity of kernel

Computer architecture is shifting towards less and less Byte/flop(BF) ratio. A summary of BF ratio of various architecture from each vendor is shown in Table 4.1 [26].

As pointed out in [26], It is quite common that algorithms with low BF ratio (dense linear algebra) have high complexity O(N^3), and algorithms with low complexity (FFT, sparse linear algebra) have high BF ratio. The FMM [27]has an exceptional combination of O(N) complexity and a BF ratio that is even lower than matrix-matrix multiplication. We took simple code of matrix multiplication over gpu from 'Polybench kernels implementation on CUDA' and try to analyse with BF ratio while changing its loop ordering. The DL_{total} of each of the loop ordering is shown in 4.2.

Listing 4.1: Modified matrix multiplication code

```
for(int I=0;I<=32;I++)
    for(J=0;J<=64;J++)
        for(K=0;K<=128;K++)
            C[I*64 +J] = C[I*64+J] + A[I*128 +K]*B[K*64 +J];
```

For Table 4.2 cache line size is 16 elements along with cache of size 256 elements. Hence when the count DL_{total}[section 3.8] count reaches above 2^{12}(4096) elements, we consider cache to be flushed and hence reuse after that loop will not take place. This has been denoted by 'Over' which shows that loop has overflown and hence DL_{total} is now simply multiplied by the number of run(upper bound - lower bound) of that loop and all the outer loops.

Array	I	J	K	IJ	IK	JI	KI	KJ	KJ
C	32	4	1	132	32	132	4	32	4
A	32	1	8	32	264	32	8	264	8
B	1	64	8	64	8	64	264	8	264
C+A+B	65	69	17	228	304	228	276	304	276
Table continued									
Array	IJK	IKJ	JIK	JKI	KIJ	KJI			
C	132	Over	132	Over	Over	Over			
A	264	Over	264	Over	Over	Over			
B	264	Over	264	Over	Over	Over			
C+A+B	29184	19456	29184	8832	19456	8832			

Table 4.2: Cache size of 2^{12} elements i.e total 256 cache lines

From the table 4.2 based on just DL_{total} we see that 'KJI' loop ordering will

be maximally cache effective on a system with cache line size and cache size of 16 and 256 elements respectively. While If we take the same system with increased cache size of $2^{14}(16384)$ elements then it turns out that any loop ordering will result into same DL_{total}. This result is shown in table 4.3 this happens since the cache size is big enough to accommodate all the elements hence every element will be brought once only to cache. Interestingly, the BF ratio of the kernel will then be $660 * 16/(32 * 64 * 128) = 0.003$ which if compared from table 4.1 is smaller than any available modern microprocessors. Hence this kernel would be compute bound in all of them which means that with 2^{14} elements cache size no further improvement can be done in kernel of 4.1, since data cannot be processed as fast as it is supplied to the processor.

Array	I	J	K	IJ	IK	JI	KI	KJ	KJ
C	32	4	1	132	32	132	4	32	4
A	32	1	8	32	264	32	8	264	8
B	1	64	8	64	8	64	264	8	264
C+A+B	65	69	17	228	304	228	276	304	276
Table continued									
Array	IJK	IKJ	JIK	JKI	KIJ	KJI			
C	132	132	132	132	132	132			
A	264	264	264	264	264	264			
B	264	264	264	264	264	264			
C+A+B	660	660	660	660	660	660			

Table 4.3: Cache size of 2^{14} elements i.e total 1024 cache lines

Chapter 5

Results and Conclusion

This chapter discusses details about our results and the conclusion that we achieve. First section discusses the design and implementation of the system. We have used Cetus compiler to achieve results on standard benchmarks. We will use same benchmark program over different cache sizes. We provide insight into what could possibly be future work in this direction and how we can evaluate other transformations based on this model.

5.1 Design and Implementation

We have modified cetus compiler[28] to include one more pass. It now first checks the syntax of the code and then calculates bf ratio of the code and then decides which loop permutation will be good for a particular system parameters. To better understand our results we are using cache of three different sizes and then compare our results with the results obtained on the simulation of these programs with the same set of parameters.

5.2 Testing setup

We have assumed the loops to be properly nested with no conditional statements. For this, we have taken portion of code from standard benchmarks5.3 that are

following our input constraints and we output the best possible permutation of loop such that the program is better in cache effectiveness. To show effectiveness of our algorithm 3.3 we have modified the default values of some variables. We refer to these modified programs with "_mod" as the suffix to the name of the program. We also assume code fragment to be completely nested hence we have removed all the loop invariant transformation, since this does not change meaning of the code. All the experiment has been done on Cetus compiler [28]. This is a source to source compiler which is written in JAVA. The simplicity of the language without pointers and user defined types makes the compiler source code fairly easy to understand. The design of the compiler is extensible to support multiple languages which make it really flexible along with being easily operable.

5.3 Standard Benchmarks

For initial results, we took codes from polybench benchmark [29] which contains collection of polybench codes as well as convolutional codes implemented for cuda and other programing languages. Code from the following function of polybench benchmark were used to get the final results:

1. code from $mm3\_cpu()$ function in filename 3mm.cu
2. code from $init()$ function in filename 3DConvolution.cu
3. code from $runMvt()$ function in filename mvt.cu
4. code from $syrk()$ function in filename syrk.cu

5.4 Performance Results

In this section, we have taken one program from four different sections of benchmarks. Each of them is then modified to double the available programs. The results of all these codes is merged in the last table. Finally, we show how our method helps

in counting precise Distinct Lines count and how it used to calculate B/F-Ratio through which we can classify program as memory bound or compute bound.

5.4.1 Matrix Multiplication

Listing 5.1: Matrix Multiplication

```
for(int I=0; I<NI; I++){
    for(int J=0; j<NJ; J++){
        for(int K=0; K<NK; K++){
            E[I*NJ+J] = E[I*NJ+J] + A[I*NK+K] * B[K*NJ+J];
        }
    }
}
```

Table 5.1 shows the results obtained for code listing 5.2. The column cache size denotes the total cache lines present in the cache, while column block size signifies the total element in a single cache line. In this chapter we will use block size and line size interchangeably, both denoting to the cache line size. From Table 5.1 we can infer that keeping any loop permutation will not affect cache performance.

Cache Size	Block Size	Array	I	J	K	IJ	IK	JI	JK	KI	KJ
512	2^8	E	512	2	1	Over	Over	Over	Over	Over	Over
		A	512	1	2	Over	Over	Over	Over	Over	Over
		B	1	2	512	Over	Over	Over	Over	Over	Over
		E+A + B	515	515	515	515 * 512	515 * 512	515 * 512	515 * 512	515 * 512	515 * 512
Table continued											
Cache Size	Block Size	Array	IJK	IKJ	JIK	JKI	KIJ	KJI			
512	2^8	E	Over	Over	Over	Over	Over	Over			
		A	Over	Over	Over	Over	Over	Over			
		B	Over	Over	Over	Over	Over	Over			
		E+A + B	515 * 512^2	515 * 512^2	515 * 512^2	515 * 512^2	515 * 512^2	515 * 512^2			

Table 5.1: Matrix multiplication code running with cache size of 512 lines and block size of 128 elements

Consider another system with larger cache size and less block size which means it can accumulate more lines in the cache. Table 5.2 shows the result when cache size of 2048 cache lines with each cache line of 256 elements is used. The overall DL count (distinct lines count) of the program has decreased, but still any permutation of the loop is favourable.

Cache Size	Block Size	Array	I	J	K	IJ	IK	JI	JK	KI	KJ
2048 2^{11}	2^8	E	512	2	1	1024	512	1024	2	512	2
		A	512	1	2	512	1024	512	2	1024	2
		B	1	2	512	2	512	2	1024	512	1024
		E+A + B	515	515	515	1538	2048	1538	1028	2048	1028
Table continued											
Cache Size	Block Size	Array	IJK	IKJ	JIK	JKI	KIJ	KJI			
2048 2^{11}	2^8	C	1024	1024	1024	1024	1024	1024			
		A	1024	1024	1024	1024	1024	1024			
		B	1024	1024	1024	1024	1024	1024			
		E+A + B	3072	3072	3072	3072	3072	3072			

Table 5.2: Matrix multiplication with increased cache size of 2048 cache lines

Consider code listing 5.2 which is slightly modified version of the matrix multiplication. Although the semantic of the program has changed but we have introduced only constant multiples thus maintaining the structure and syntax. This will help us display how cache line and block size can affect favourable loop permutation.

Listing 5.2: Modified matrix multiplication

```
for(int I=0;I<320;I++){
    for(int J=0;j<640;J++){
        for(int K=0;K<1280;K++){
            E[I*64+J*99] = E[I*64+J*99] + A[I*128+K*151]*B[K*64+J*5];
        }
    }
}
```

From the table 5.3, it can be inferred that with cache size of 8192 lines and block size of 16 elements, cache overflow first takes place if IK ordering is used. After Over(overflow) the DL count till then will be simply multiplied by the number of times that loops execute and the number of times its containing loops execute. If the minimum value is zero then we have to simply multiply it with upper bound of loop variable. It can be easily seen from table 5.3 that IJK is the most favourable combination with I being the innermost loop and K being the outermost loop.

Cache Size	Block Size	Array	I	J	K	IJ	IK	JI	JK	KI	KJ
8192 2^{13}	2^4	E	320	7	1	132	32	132	4	32	4
		A	320	1	1280	320	Over	320	1280	Over	1280
		B	1	40	1280	5316	1280	40	5316	1280	5316
		E+A + B	641	48	2561	5768	410592	492	6600	410912	6600
Table continued											
Cache Size	Block Size	Array	IJK	IKJ	JIK	JKI	KIJ	KJI			
8192 2^{13}	2^4	E	132	Over	Over	Over	Over	Over			
		A	Over	Over	Over	Over	Over	Over			
		B	5316	Over	Over	Over	Over	Over			
		E+A + B	415048	410592 x 640	629760	2112 x1000	410912 x640	2112000			

Table 5.3: Table showing results of code listing 5.2

5.4.2 3DConvolution

Convolution is widely used and hence comes with polybench benchmark. The code from the *init*() function of the file 3Dconvolution.cu is presented in code listing 5.3. It basically initialises Array A with appropriate values which are further used in 3D convolution. The default values of NI ,NJ ,NK , P are 256 , 256 , 256 , 1.

Listing 5.3: init() function of 3Dconvolution.cu

```
for (I = 0; I < NI; ++I){
    for (J = 0; J < NJ; ++J){
        for (K = 0; K < NK; ++K){
            A[I*(NK * NJ) + J*NK + P*K] = I \% 12 + 2 * (J \% 7) + 3 * (K \% 13);
        }
    }
 }
```

The result of code listing 5.3 is shown in the following table 5.4. It can be seen that KIJ (K being the innermost and then I or J can be in any order) is the most favourable loop ordering. Although the cache is big enough to contain all the code still it is favourable to keep K as innermost since it uses minimum number of cache lines.

Cache Size	Block Size	Array	I	J	K	IJ	IK	JI	JK	KI	KJ
2^{18}	2^5	A	256	256	8	65536	2048	65536	2048	2048	2048
Table continued											
Cache Size	Block Size	Array	IJK	IKJ	JIK	JKI	KIJ	KJI			
2^{18}	2^5	A	65536	65536	65536	65536	65536	65536			

Table 5.4: Table showing result of code listing 5.3

We slightly modify the init() function as shown in code listing 5.4. The result of this modified code can be seen in the following table 5.5.

Listing 5.4: init_mod of 3DConvolution

```
for (I = 0; I < 101; ++I){
    for (J = 0; J < 201; ++J){
        for (K = 0; K < 301; ++K){
            A[I*47 + J*51 + K*28] = I \% 12 + 2 * (J \% 7) + 3 * (K \% 13);
        }
    }
 }
```

It can easily seen from the table 5.5 below, IKJ(I being the innermost) is the most favourable permutation of loops. All the values are in unit cache lines except the cache size and block size which are in unit lines and unit elements. Over in every table denotes that cache has overflown and hence DL count will be simply multiplied by upper bound of that and all its containing loops(since lower bound of loop variable is 0).

Cache Size	Block Size	Array	I	J	K	IJ	IK	JI	JK	KI	KJ
2^{12}	2^2	A	100	200	300	3599	3144	3599	4566	3144	4566
Table continued											
Cache Size	Block Size	Array	IJK	IKJ	JIK	JKI	KIJ	KJI			
2^{12}	2^2	A	Over	Over	Over	Over	Over	Over			

Table 5.5: Table showing result of code listing 5.4

5.4.3 MVT

MVT stands for Matrix Vector Product and Transpose which has been part of polybench benchmark from version 1.0. It can be found under kernels in linear algebra section. Following code listing 5.5 is code fragment from $runMvt()$ function.

Listing 5.5: runMvt() from MVT

```
for (I = 0; I < 4096; ++I){
    for (J = 0; J < 4096; J++){
            X[I] = X[1] + A[I*4096 + J] * Y[J];
    }
 }
```

From the table 5.6 it can be seen that with cache size of 2^{13} lines and block size of 2^2 elements, JI is cache effective permutation, with J being the innermost loop. Since it takes only 2049 lines while I as the innermost loop takes 5151 lines.

We then modify this code from runMvt() function is shown in listing 5.6. The upper bounds of both the loop and constant multiplied with I and J in array index of array A is changed.

Cache Size	Block Size	Array	I	J	IJ	JI
2^{13}	2^2	A	2^{12}	2^{10}	Over	Over
		X	2^{10}	1	2^{10}	2^{10}
		Y	1	2^{10}	2^{10}	2^{10}
		A+X + Y	5121	2049	$2^{22} + 2^{11}$	$2^{22} + 2^{11}$

Table 5.6: Table showing result of code listing 5.5

Listing 5.6: Modified code of runMvt

```
for (I = 0; I < 201; ++I){
    for (J = 0; J < 501; J++){
            X[I] = X[1] + A[I*51 + J*90] * Y[J];
    }
 }
```

From the following Table 5.7 it is evident that IJ loop ordering is more cache effective in this case. While this is complete opposite for what we obtained for original code listing 5.5.

Cache Size	Block Size	Array	I	J	IJ	JI
2^9	2^2	A	200	500	Over	Over
		X	50	1	50	50
		Y	1	50	50	50
		A+X + Y	251	551	13598	13598

Table 5.7: Table showing result of modified runMvt

5.4.4 SYRK

SYRK stand from Symmetric rank-k operations. It also a kernel in linear algebra section of polybench benchmark. Following code listing 5.7 is from the $syrk()$ function. The default values of N, M and P are 1024 , 1024 and 1.

Listing 5.7: syrk() from SYRK

```
for (I = 0; I < N; I++){
    for (J = 0; J < N; J++){
        for (K = 0; K < M; K++){
            C[I*N + J] += alpha * A[I*M + P*K] * A[J*M + P*K];
        }
    }
}
```

The following table 5.8 shows that K should be innermost loop with cache size of 2^{12} and block size of 2^2. The cache will first overflow at IJ and JI indicating that none of them can be innermost loop.

Cache Size	Block Size	Array	I	J	K	IJ	IK	JI	JK	KI	KJ
2^{12}	2^2	C	1024	256	1	Over	1024	Over	256	1024	256
		A	1025	1025	256	1024	Over	1024	Over	Over	Over
		C+A	2049	1281	257	$2^{10} + 2^{18}$	$2^{10} + 2^{18}$	$2^{10} + 2^{12}$	$2^8 + 2^{18}$	$2^{10} + 2^{18}$	$2^8 + 2^{18}$
Table continued											
Cache Size	Block Size	Array	IJK	IKJ	JIK	JKI	KIJ	KJI			
2^{12}	2^2	C	Over	Over	Over	Over	Over	Over			
		A	Over	Over	Over	Over	Over	Over			
		C+A	$2^{18} + 2^{20}$	$2^{18} + 2^{20}$	$2^{18} + 2^{20}$	$2^{18} + 2^{20}$	$2^{18} + 2^{20}$	$2^{18} + 2^{20}$			

Table 5.8: Table showing result of code listing 5.7

We slightly modify the code of *syrk*() as shown in 5.8. the upper bound of all the loop variable and the constant multiplied with loop variable in the Array index of Array A is changed. The default values of N, M and P are 1024 , 1024 and 1 and alpha is a constant.

Listing 5.8: Modified syrk() from SYRK

```
for (I = 0; I < 128; I++){
    for (J = 0; J < 256; J++){
        for (K = 0; K < 512; K++){
            C[I*256 + J] += alpha * A[I*64 + K*97] * A[J*64 + K*97];
        }
    }
}
```

The result of code listing 5.8 can be seen in the following table 5.9. The table shown that I should be innermost loop while K should be outermost loop. hence, IJK is most cache effective loop ordering.

Cache Size	Block Size	Array	I	J	K	IJ	IK	JI	JK	KI	KJ
10^4	2^2	C	128	64	1	8192	128	8192	64	128	64
		A	128	256	128	128	Over	128	Over	Over	Over
		C+A	256	320	129	8320	$2^7 + 2^{16}$	8320	$2^6 + 2^{17}$	$2^7 + 2^{14}$	$2^6 + 2^{15}$
Table continued											
Cache Size	Block Size	Array	IJK	IKJ	JIK	JKI	KIJ	KJI			
10^4	2^2	C	8192	8192	8192	8192	8192	8192			
		A	Over	Over	Over	Over	Over	Over			
		C+A	$2^{13} + 2^{14}$	$2^{13} + 2^{14}$	$2^{13} + 2^{14}$	$2^{13} + 2^{14}$	$2^{13} + 2^{14}$	$2^{13} + 2^{14}$			

Table 5.9: Table showing results of code listing 5.8

If in the same code we just decrease the cache size while keeping everything else constant we will get different result, which is shown in the following table 5.10.

We now show how we are able to count the Distinct Cache lines of code more precisely than existing approaches. We calculate the B/F ratio of each code each with different cache size as shown in the result table 5.11. For machine B/F-Ratio, we consider Intel Sandy Bridge with B/F-Ratio of 0.211. Hence any program having B/F-Ratio less than 0.211 will be classified as compute bound while programs having B/F-Ratio higher than 0.211 will be memory bound program on this particular

Cache Size	Block Size	Array	I	J	K	IJ	IK	JI	JK	KI	KJ
2^{15}	2^2	C	128	64	1	8192	128	8192	64	128	64
		A	128	256	128	128	13736	128	13736	13736	13736
		C+A	256	320	129	8320	13864	8320	13800	13864	13800
Table continued											
Cache Size	Block Size	Array	IJK	IKJ	JIK	JKI	KIJ	KJI			
2^{15}	2^2	C	8192	8192	8192	8192	8192	8192			
		A	15784	15784	15784	15784	15784	15784			
		C+A	23976	23976	23976	23976	23976	23976			

Table 5.10: Table showing result of code listing 5.8

system. Consider the following table 5.11, block Size is the total number of elements in one cache line while DL Count is the number of distinct cache lines being used by that particular array. The Cache Size is considered to be large enough such that it does not overflow in any case.

Name	Block Size	Array	DL Count (Actual)	DL Count (Previous)	DL Count (Our Method)
mm3_cpu	2^8	A	1024	1024	1024
mm3_cpu_mod	2^4	A	20003	20071	20003
init	2^2	A	65536	65536	65536
init_mod	2^2	A	5789	5825	5789
runMvt	2^2	X	4096	4096	4096
runMvt_mod	2^2	X	13598	13650	13598
syrk	2^2	A	262144	262144	262144
syrk_mod	2^2	A	13736	14424	13736

Table 5.11: Comparison of distinct lines(DL) count by our and previous approach

As discussed earlier in chapter 4, if the B/F-Ratio of the program is larger than machine B/F-Ratio then it is Memory bound otherwise it is compute bound. The result of all the 4 different programs matrix multiplication, 3DConvolution, MVT, SYRK along with their modified versions, is shown in table 5.12. Here Block Size is the total number of elements in one cache line and while DL Count and cache size is the number of distinct cache lines being used by that particular array and total number of lines present in cache. FLOPS here denoted total floating point being done in the entire loop nest. We consider Intel Sandy bridge with bfratio 0.2111 as

given in table 5.12, hence any program having bfratio greater than this is memory bound otherwise compound bound.

Name	Cache Size	Block Size	Total FLOPS	DL Count	B/F-Ratio	Remark
mm3_cpu	512	2^8	$2*512^3$	$515*512^2$	2^7	Memory Bound
mm3_cpu_mod1	2^{11}	2^8	$2*512^3$	3072	0.002	Compute Bound
mm3_cpu_mod2	2^{13}	2^4	$2*512^3$	415048	0.024	Compute Bound
init	2^{18}	2^5	$4*256^3$	65536	0.31	Compute Bound
init_mod	2^{12}	2^2	4*100 *200 *300	5789	0.0009	Compute Bound
runMvt	2^{13}	2^2	$2*4096^2$	4196352	0.5002	Memory Bound
runMvt_mod	2^9	2^2	2*500*200	13598	0.271	Memory Bound
syrk	2^{12}	2^2	$3*1024^3$	1310720	0.0016	Compute Bound
syrk_mod1	10^4	2^2	3*128 *256 *512	24576	0.0004	Compute Bound
syrk_mod2	2^{15}	2^2	3*128 *256 *512	23976	0.0019	Compute Bound

Table 5.12: table showing B/F-Ratio of different kernels

5.5 Conclusion and Future Work

We calculate B/F Ratio of the programs and then decide whether it is profitable to do loop interchange or not. We classify the programs into memory bound and compute bound. If the program is memory bound on a particular specification then it calculates the best possible loop ordering which is optimal for system cache. This is done with the help of finding distinct cache lines being used by the program with the help of DL Model as presented in chapter 3. We have further improved this DL Model such that now it is more precise than earlier. We finally show our results on programs of polybench benchmarks and found that majority of them were compute bound.

Various challenges occurred during the implementation of this concept due to which we still have some limitation in our system like

- We only deal with properly nested loops since performing loop interchange over other types of loops is generally not feasible.
- We ignore conditional statements (if any) in the program and consider that conditional statements will execute every time
- All the array index should be linear combination of loop variable while we plan to extend this work to nonlinear combination but it is still a challenge for us.
- We plan to extend this concept of B/F Ratio so that it can also guide other loop transformations such as deciding tile size, loop fusion etc.

References

[1] Vivek Sarkar. "Determining average program execution times and their variance". In: *ACM SIGPLAN Notices*. Vol. 24. 7. ACM. 1989, pp. 298–312.

[2] Jeanne Ferrante, Vivek Sarkar, and W. Thrash. "On Estimating and Enhancing Cache Effectiveness". In: *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*. London, UK, UK: Springer-Verlag, 1992, pp. 328–343. ISBN: 3-540-55422-X. URL: http://dl.acm.org/citation.cfm?id=645669.665222.

[3] Michael Wolfe. "Optimizing supercompilers for supercomputers". In: (1989).

[4] Allan Kennedy Porterfield. "Software Methods for Improvement of Cache Performance on Supercomputer Applications". AAI9012855. PhD thesis. Houston, TX, USA, 1989.

[5] Zhiyu Shen, Zhiyuan Li, and PEN-CH YEW. "An empirical study on array subscripts and data dependencies". In: *1989 International Conference on Parallel Processing, University Park, PA*. 1989.

[6] Vivek Sarkar. "Automatic selection of high-order transformations in the IBM XL FORTRAN compilers". In: *IBM Journal of Research and Development* 41.3 (1997), pp. 233–264.

[7] Jun Shirako and Vivek Sarkar. "Oil and Water can mix! Experiences with integrating Polyhedral and AST-based Transformations". In: ().

[8] Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J Ramanujam, P Sadayappan, and Vivek Sarkar. "Analytical bounds for optimal tile size selection". In: *Compiler Construction*. Springer. 2012, pp. 101–121.

[9] Ken Kennedy and Kathryn S McKinley. "Optimizing for parallelism and data locality". In: *25th Anniversary International Conference on Supercomputing Anniversary Volume*. ACM. 2014, pp. 151–162.

[10] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. "Cache miss equations: An analytical representation of cache misses". In: *Proceedings of the 11th international conference on Supercomputing*. ACM. 1997, pp. 317–324.

[11] Amitabha Tripathi. "ON A LINEAR DIOPHANTINE PROBLEM OF FROBENIUS". In: *INTEGERS: ELECTRONIC JOURNAL OF COMBINATORIAL NUMBER THEORY* 6.A14 (2006), A14.

[12] JANET TRIMM. "On Frobenius numbers in three variables". In: (2006).

[13] Jorge L. Ramírez Alfonsín. *The Diophantine Frobenius Problem*. Oxford University Press, 2006. ISBN: 978-0-19-856820-9.

[14] Alfred Brauer and James E Shockley. "On a problem of Frobenius". In: *J. reine angew. Math* 211 (1962), pp. 215–220.

[15] S.M. Johnson. "A Linear Diophantine Problem". In: *Canadian Journal of Mathematics* (1960), pp. 390–398.

[16] Öystein J Rödseth. "On a linear Diophantine problem of Frobenius." In: *Journal für die reine und angewandte Mathematik* 301 (1978), pp. 171–178.

[17] Michael E. Wolf and Monica S. Lam. "A Data Locality Optimizing Algorithm". In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI '91. Toronto, Ontario, Canada: ACM, 1991, pp. 30–44. ISBN: 0-89791-428-7. DOI: 10.1145/113445.113449. URL: http://doi.acm.org/10.1145/113445.113449.

[18] David Callahan and Allan Porterfield. "Data Cache Performance of Supercomputer Applications". In: *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. Supercomputing '90. New York, New York, USA: IEEE Computer Society Press, 1990, pp. 564–572. ISBN: 0-89791-412-0. URL: http://dl.acm.org/citation.cfm?id=110382.110587.

[19] Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Norwell, MA, USA: Kluwer Academic Publishers, 1988. ISBN: 0898382890.

[20] K. Gallivan, W. Jalby, and D. Gannon. "On the Problem of Optimizing Data Transfers for Complex Memory Systems". In: *Proceedings of the 2Nd International Conference on Supercomputing*. ICS '88. St. Malo, France: ACM, 1988, pp. 238–253. ISBN: 0-89791-272-1. DOI: 10.1145/55364.55388. URL: http://doi.acm.org/10.1145/55364.55388.

[21] Naznin Fauzia, Venmugil Elango, Mahesh Ravishankar, J Ramanujam, Fabrice Rastello, Atanas Rountev, Louis-Noël Pouchet, and P Sadayappan. "Beyond reuse distance analysis: Dynamic analysis for characterization of data locality potential". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 10.4 (2013), p. 53.

[22] Lorena A Barba and Rio Yokota. "How will the fast multipole method fare in the exascale era". In: *SIAM News* 46.6 (2013), pp. 1–3.

[23] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*. Benjamin/Cummings Publishing Company Redwood City, CA, 1994.

[24] Steve Carr and Ken Kennedy. "Improving the ratio of memory operations to floating-point operations in loops". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.6 (1994), pp. 1768–1810.

[25] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[26] Rio Yokota. "An FMM based on dual tree traversal for many-core architectures". In: *Journal of Algorithms & Computational Technology* 7.3 (2013), pp. 301–324.

[27] Nader Engheta, William D Murphy, Vladimir Rokhlin, and Marius S Vassiliou. "The fast multipole method (FMM) for electromagnetic scattering problems". In: *Antennas and Propagation, IEEE Transactions on* 40.6 (1992), pp. 634–641.

[28] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. "Cetus: A source-to-source compiler infrastructure for multicores". In: *Computer* 42.12 (2009), pp. 36–42.

[29] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. "Auto-tuning a high-level language targeted to GPU codes". In: *Innovative Parallel Computing (InPar), 2012*. IEEE. 2012, pp. 1–10.

[30] Sang-Ik Lee, Troy A Johnson, and Rudolf Eigenmann. "Cetus–an extensible compiler infrastructure for source-to-source transformation". In: *Languages and Compilers for Parallel Computing*. Springer, 2004, pp. 539–553.