# Optimizing Array Locality via Memory Layout Reorganization

Xuezhi Wang (xuezhiw@cs.cmu.edu), Junchen Jiang (junchenj@cs.cmu.edu)

May 1, 2013

## 1 Introduction

### 1.1 Problem

When optimizing array locality for cache performance in a loop, a conventional approach is tiling which keeps the current memory layout of array but change execution order of instructions so that cached content will be reused for future execution before they are replaced. However, this scheme may fail when the execution reordering is unavailable due to data dependency or other constraints.

### 1.2 Our approach

In this project, we propose an alternative approach for optimizing array locality through reorganizing memory layout of arrays. The main idea is that by analyzing the access pattern of array elements, we can reorganize the array memory layout such that the new memory layout can achieve a better locality even with no changes on execution order or prefetching (though it can potentially be used together with tiling to further improve the performance).
A simple example of using the new method is the following:

```
for(int i=0; i<n; i++){
   A[i][1] += 1;
   A[i][0] += 2;
}
```

In this program, assuming cache can hold 2 elements, a naive execution (no tiling) will cause cache miss on every access. In our method, we can reorganize the 2d array A[n][2] into 1-d array B[2n] such that B[0]=A[0][1], B[1]=A[0][0], B[2]=A[1][1], B[3]=A[1][0], and the cache miss rate will be reduced to half (100% to 50%) of that in a naive execution with no change on the order of execution.
In case there are multiple access patterns (e.g., multiple loops for the same array), we

consider the following two options:
(1) For each access pattern, create a suitable memory layout and keep this newest layout(s) (keep multiple copies only when necessary, e.g. for branches);
(2) Find the most suitable memory layout for all access patterns, and hopefully it should be better than the original layout (which does not take access patterns into consideration). It is even possible that which option is better is dynamically decided by the access patterns in the code.
The key difference between tiling and our method is, tiling changes the execution order of instructions, but our method changes the memory layout and can also be used in the case that instruction order should be fixed during program execution. Certainly our method can be used in the cases where tiling is used. Another example where our method might be more useful would be, in a program involving regular expression matching, if we use an automaton to store the transition states, the order of state transition in actual execution cannot be changed (i.e., tiling is not applicable), but we can reorganize the order of state storage by analyzing the degree/transition relations of each state.

## 1.3 Related Work

Starting from [1], many related works have been published on tiling optimization for loops. Recent developments mainly focus on tiling size selection (e.g., [3, 4]) and specific hardware support for blocked optimization (e.g., [2]). There are also works on optimization of loop performance using hardware support without tiling (e.g., [5]). Another paradigm is through exploiting array and loop parallelism through dynamic analysis (e.g., [6]) and speculative execution(e.g., [7]).

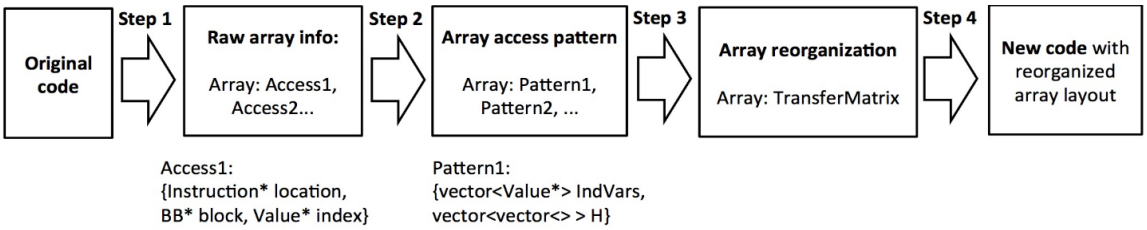
## 1.4 Contribution

- We present a general framework for optimizing array locality by memory layout reorganization.
- We propose a novel algorithm to determine a better memory layout based on access pattern analysis.
- We extend our algorithm to handle multiple access patterns.
- We use cache simulators (PIN tools) to measure the cache performance before and after applying our algorithm. Experimental results have shown that our algorithm can achieve significant improvement on cache performance.

# 2 Approach

## 2.1 Framework

First we present a general framework for memory layout reorganization as demonstrated by the figure below:




The core concept of the framework is *access pattern*. For each array in the program, let each use of the array (including load and store) be an *access*. An access is a wrapped class that describes the location (including the instruction and basic block), dimension information (e.g., number of dimension, size of each dimension) index formulation (i.e., relationship to each indection variable), and loop information (e.g., number of iteration) of the access. Then the access pattern of an array is defined to be a class to serve two purposes:

- Summarize the useful information of all accesses on the array, and
- Provide functionalities to make layout reorganization decision.

The most critical functionalities of an access pattern is to summarize each access as an induction variable matrix ($H$), and to calculate the new layout decision represented by a transfer matrix ($P$) between old layout and the new one.
Our optimization system is comprised of four steps that take as input an original C code and generate an optimized LLVM object code. We now present the design and implementation.

- Step 1 makes a pass on the original code to get access information for each array. The program goes through each instruction and identify the initialization of an array by checking its class and analyze it to find the dimension information and data type that it allocates for. For each array found, an access pattern class will be created to store all accesses. The program will then be scanned again to find all access location and information.

- Step 2 summarizes the raw information of multiple access for each access pattern. The key results include induction variable matrix ($H$) and the weight of each access. Each row of the induction variable matrix consists of the coefficients of each induction variable to calculate one index value of the access. For example,

each access to a two-dimensional array within a three level nested loop can be represented by an induction variable matrix with two rows and three columns. The weight for each access is a rough estimation on the number of access. For now, it simply multiplies the range of all induction variables and does not involve dynamic analysis for conditional branches that may change number of access in runtime. It provides a way to balance between multiple accesses.

- Step 3 calculates a reorganization for each array based on the access patterns. The reorganization is essentially a linear transformation (through a transfer matrix $P$) between index of original array and that of the new array. Through the transfer matrix $P$, the access on the original array layout will be mapped to the new array layout in hope that the access for the innermost loop can be within the same dimension and with less stride on the element. We will formally defined it as a linear optimization problem in the next section. Due to the inherant difficulty of the problem, it is not guaranteed to find a feasible solution for the transfer matrix. In case a transfer matrix is not found, we will back-off to original layout (i.e., an identical matrix). Besides the transfer matrix, this step also generate the new size of each dimension of the new layout, so that when accessing index on the new array, it will not go out of bound.

- Step 4 generates the optimized code by initializing a new array with new size for each old array and replacing all accesses to the old array to the corresponding new array with index calculated by the transfer matrix. To redirect all use of an array to its new layout, we first add replace all accesses with new access and remove the initialization of the old array for safety. Finally, a dead-code-elimination pass is performed to clean up all use and def of the old arrays.

## 2.2 Core algorithm

The key part in this framework is how to determine a better memory layout based on the information gathered thtough access pattern analysis. We propose a novel algorithm as follows:
Consider an array $A$ with $m$ dimension used in a $n$-level nested loop (induction variable $i_1, ..., i_n$ from outmost loop to innermost loop), and $[x_1 \ ... \ x_m]^\top = H[i_1 \ ...i_n]^\top$, where $H$ is an $m \times n$ induction variable transform matrix (given by the analysis on the access patterns). For a new array $B$ with dimension $d$, where the correspondence between $A[x_1]...[x_m]$ and $B[x'_1]...[x'_d]$ is given by:

$$[x'_1 \ ... \ x'_d]^\top = P[x_1 \ ... \ x_m]^\top = PH[i_1 \ ... \ i_n]^\top$$

where $P$ is denoted as the TransferMatrix with size $d \times m$.
If each induction variable $x_i$ increments by $\delta_i$, we have the corresponding transforma-

tion:

$$[x'_1 + \delta'_1 \ ... \ x'_d + \delta'_d]^\top = PH[i_1 + \delta_1 \ ... \ i_n + \delta_n]^\top$$

which gives:

$$[\delta'_1 \ ... \ \delta'_d]^\top = PH[\delta_1 \ ... \ \delta_n]^\top \quad (1)$$

Hence the key idea of our algorithm is:
*Find a TransferMatrix $P$ to achieve: when each induction variable $x_i$ increments by $\delta_i$, if we access the corresponding elements in $B$ rather than $A$, we could have much better locality, i.e., those $\delta'_1, ..., \delta'_d$ given by Eq. 1 satisfy:*

$$|\delta'_1| + ...|\delta'_d| \leq C(|\delta_1| + ... + |\delta_n|)$$

In other words, when the induction variables have a small change, even it does not necessarily result in a small change in the indices accessed in the original array, it should result in a relatively small change in the indices accessed in the reorganized array.
In practical, given that the innermost loop is the most costly one, we can set $\delta_1, ..., \delta_{n-1}$ and $\delta'_1, ..., \delta'_{n-1}$ to zero and further simplify the condition as:

$$|\delta'_d| \leq C|\delta_n|$$

or in a more formal way: when the innermost induction variable increments by $\delta_n$, find the matrix $P$ that satisfy:

$$\min_P \delta'_d \quad \text{subject to}$$
$$[0 \ ... \ \delta'_d]^\top = PH[0 \ ... \ \delta_n]^\top$$

We can solve this equation using integer programming methods, and use the resulting $P$ matrix to transform the original code.

## 2.3 Boundary Issues

- **Size of the new Array** $B$. When we apply the above algorithm for array transformation, we need to assign new sizes to the new array. The new sizes should satisfy the constraint that when we visit the new indices we will not run out of bound, and also they are not too large since we do not want to put high pressure on memory allocation. During the process of analyzing access patterns, we record the upper and lower bound of the accessed indices, which we used with transformation matrix to compute the new upper/lower bound of indices.

- **Uniqueness of mapping**. When we transform the array we are effectively computing a mapping from old array to the new array. we should avoid the case that two new indices point to the same location while the old point ones point to different locations. To ensure the uniqueness of mapping we contrain the transformation matrix $P$ to have full rank.

## 2.4 Illustrative example

Suppose we want to optimize the following code by memory layout reorganization:

```
for(int i=0; i<100; i++){
    for(int j=0; j<100; j++)
        OldArray[i+2*j][3*j] = 5;
}
```

We can get the $H$ matrix by directly analyzing the code and solve for $P$ matrix by finding a feasible integer solution of the linear equations (the solution $P$ results in $\delta'_n = 1$):

$$H = \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix}, \quad P = \begin{bmatrix} 3 & -2 \\ 5 & -3 \end{bmatrix}, \text{ and } PH = \begin{bmatrix} 3 & 0 \\ 5 & 1 \end{bmatrix}$$

The IR code (just the part involving array indices) generated by the original code:

```
%2 = load i32* %j, align 4
%mul = mul nsw i32 3, %2
%3 = load i32* %i, align 4
%4 = load i32* %j, align 4
%mul4 = mul nsw i32 2, %4
%add = add nsw i32 %3, %mul4
%arrayidx = getelementptr inbounds [500 x [500 x i32]]*
                          %oldArray, i32 0, i32 %add
%arrayidx5 = getelementptr inbounds [500 x i32]*
                    %arrayidx, i32 0, i32 %mul
store i32 5, i32* %arrayidx5, align 4
br label %for.inc
```

Using the framework we proposed and the solution $P$ matrix, the corresponding IR code is transformed to the following:

```
%2 = load i32* %j, align 4
%mul = mul nsw i32 3, %2
%3 = load i32* %i, align 4
%4 = load i32* %j, align 4
%mul4 = mul nsw i32 2, %4
%add = add nsw i32 %3, %mul4
%TMP = mul i32 %mul, -3
%TMP1 = mul i32 %add, 5
%TMP2 = add i32 %TMP, %TMP1
%TMP3 = mul i32 %mul, -2
%TMP4 = mul i32 %add, 3
%TMP5 = add i32 %TMP3, %TMP4
%NewIndex = getelementptr inbounds [600 x [600 x i32]]*
```

```
                          %NewArray, i32 0, i32 %TMP5
%NewIndex6 = getelementptr inbounds [600 x i32]*
                    %NewIndex, i32 0, i32 %TMP2
store i32 5, i32* %NewIndex6, align 4
br label %for.inc
```

which is effectively equivalent to the following $C$ code:

```
for(int i=0; i<100; i++){
   for(int j=0; j<100; j++)
      NewArray[3*i][5*i+j] = 5;
}
```

From the code we can see it results in better locality of the inner loop.

### 2.5 Dealing with multiple access patterns

When there are multiple access patterns (e.g., multiple loops for the same array), if we apply our proposed algorithm naively, we can create a corresponding memory layout and keep this newest layout(s) (keep multiple copies only when necessary, e.g. for branches) for each access pattern. However in real experiments we found that it is too costly to create new arrays each time and update the correspondence. Hence we adopt the second approach, i.e., find the most suitable memory layout for all access patterns. We extend our algorithm to handle multiple access patterns as the following:
Denote the multiple access patterns in the code as $M_1, ..., M_r$ indexed by $k = 1, ..., r$. When the innermost induction variable of access pattern $M_k$ increments by $\delta_{n,k}$, find the matrix $P$ that satisfy:

$$\min_P \sum_k w_k \delta'_{d,k} \quad \text{subject to}$$
$$[0 \ ... \ \delta'_{d,k}]^\top = PH[0 \ ... \ \delta_{n,k}]^\top, \forall k$$
$$\delta'_{d,k} \geq 0, \forall k$$

where $w_k$ represents the weight given by access pattern $M_k$. In the experiments we use the number of iterations executed by access pattern $M_k$ as $w_k$. Intuitively, the larger the number of iterations executed by a certain access pattern, the higher weight we should impose on the objective function. Notice we add the nonnegative constraint $\delta'_{d,k} \geq 0$ since we assume cache works in a look-ahead way. Again we can use integer programming methods to solve the system of linear equations with constraints.

## 3 Experimental Setup

We use llvm on the Ubuntu image for generating IR code and also for transforming IR code by running function/loop passes. To measure the cache performance of the

transformed code, we use PIN tools (<www.pintool.org>) as the cache simulators and compare the following metrics:
(1) Load-Hits/Load-Misses/Load-Accesses,
(2) Store-Hits/Store-Misses/Store-Accesses,
(3) Total-Hits/Total-Misses/Total-Accesses.

To assess the performance of the proposed algorithm, we automatically generate a microbenchmark that represents different dimension array access, different levels of nested loop, and multiple access patterns. Parameters that are tunnable in generating a microbenchmark includes dimension of the array, loop depth, and induction variable matrix for each access. To test different access patterns more thoroughly, we explore all possible induction variable matrix with coefficient within 0-3, which means 256 combinations for 2-dimension with 2-level loops. We compare the performance under each configuration to evaluation the strength and weakness of our algorithm.

# 4 Experimental Evaluation

## 4.1 Results on the simple example

As an example, we run cache simulators on the test example. The cache performance results generated by PIN tools are as the following:

```
**************cache performance for original code*************
PIN:MEMLATENCIES 1.0. 0x0
#
# DCACHE stats
#
# L1 Data Cache:
# Load-Hits:                82269   98.27%
# Load-Misses:               1452    1.73%
# Load-Accesses:            83721  100.00%
#
# Store-Hits:               31725   88.24%
# Store-Misses:              4230   11.76%
# Store-Accesses:           35955  100.00%
#
# Total-Hits:              113994   95.25%
# Total-Misses:              5682    4.75%
# Total-Accesses:          119676  100.00%

**************cache performance for transformed code***********
PIN:MEMLATENCIES 1.0. 0x0
#
# DCACHE stats
#
# L1 Data Cache:
# Load-Hits:                82276   98.27%
# Load-Misses:               1446    1.73%
# Load-Accesses:            83722  100.00%
#
# Store-Hits:               34311   95.42%
# Store-Misses:              1645    4.58%
# Store-Accesses:           35956  100.00%
#
# Total-Hits:              116587   97.42%
# Total-Misses:              3091    2.58%
# Total-Accesses:          119678  100.00%
```

In the simple example we only have store instructions that rely on array locality. Hence the key metric we need to compare here is the second one: Store-Hits/Store-Misses/Store-Accesses. As we can see, the Store-Hits rate has been improved from 88.24% to 95.42%, which effectively demonstrates the resulting better locality by using our framework.

## 4.2 Results on microbenchmarks

To scale up the evaluation and test the performance in a more systematic way, we generated a microbenchmark as described in last section.
Figure 2 shows the results of Load/Store Hit Rate when dimension=2 and looplevel=2. First, in both figures, though our algorithm does not improve performance in about half the code, it can be seen that our algorithm improves the performance with more than 2% in cache hit rate for about 40% of the test code, and in best case, we can improve the performance by about 8% for Load Hit and 25% for Store Hit. Also the improvement is with discrete values which shows caching effect is not continuous. It also shows for a few number of code (about 5%), our algorithm actually reduces the cache hit rate. We attempted to dig in the reason for that, but it is challenging since PIN simulator we use works as a black box, and we believe such degradation is due to

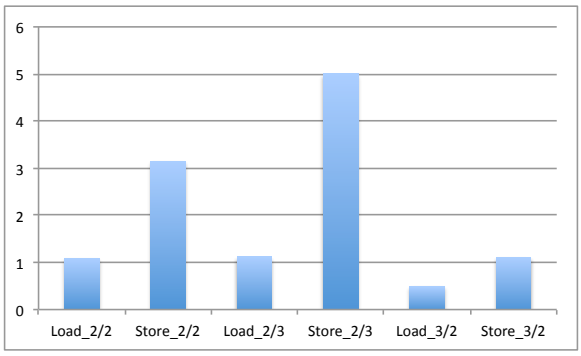



Figure 1: Bar chart of average improvements - A/B: A dimension, B loop depth.

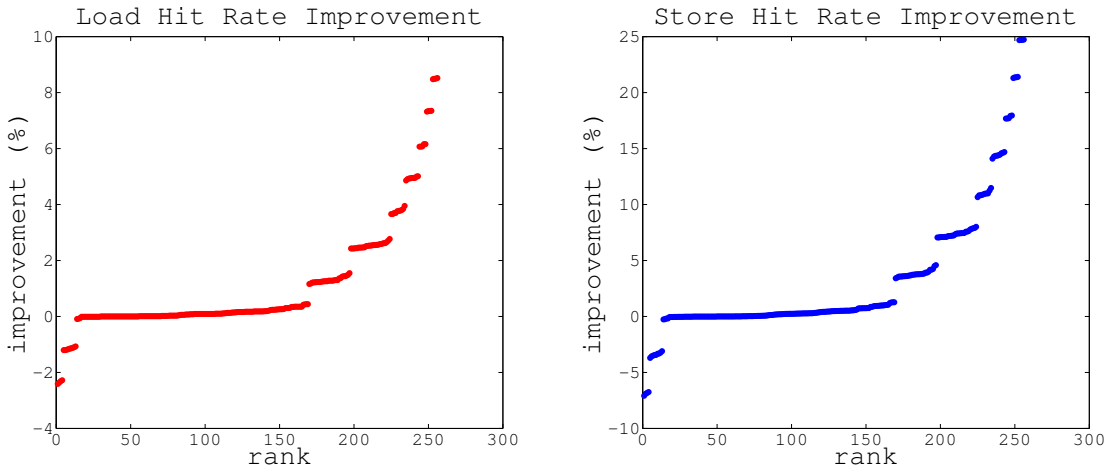



Figure 2: Load/Store Hit Rate when dimension=2 and looplevel=2

the real caching replacement policy that is more complicated than we expected.

Figure 3 shows the results of with three level loops. Firest, the result looks similar to that of Figure 2, which shows our robustness when increasing loop levels. In fact, the performance improvement in Store Hit can be much higher than Store Hit in two-level loops in its best cases, achieving about 50% improvement. We believe it is because the nested loop reduces the performance of original but our algorithm still can find the best results.

Figure 4 shows the results of with three dimension array. The improvement is much less significant in this case, but we still manage to improve for most of the code by more than 1% improvement.

Figure 5 shows the results of multiple access patterns. We compare if the two accesses are with different weight. The two accesses in the left one has equal weights (both 10K iterations), and the two accesses in the right one has very different weights (10K and 100 iterations). It shows that the improvement with two different weighted accesses has slightly higher improvement than two with equal weights. This means

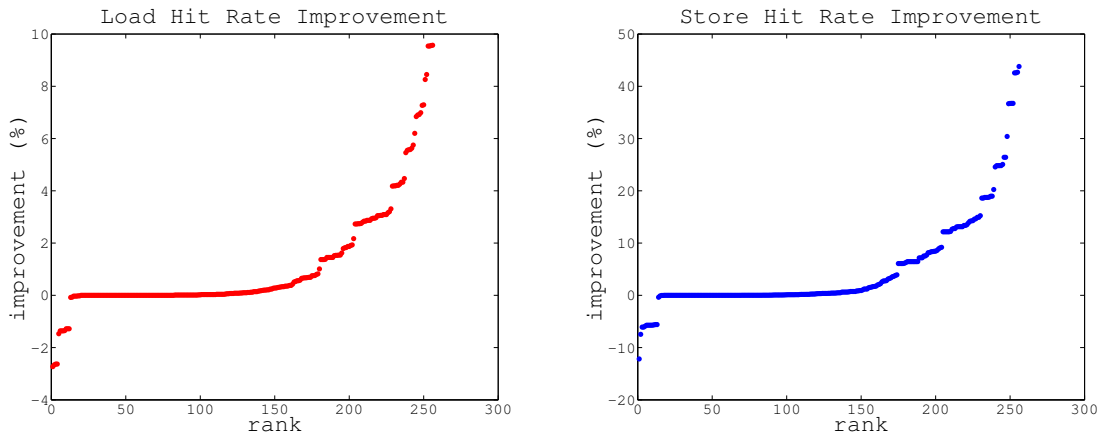



Figure 3: Load/Store Hit Rate when dimension=2 and looplevel=3

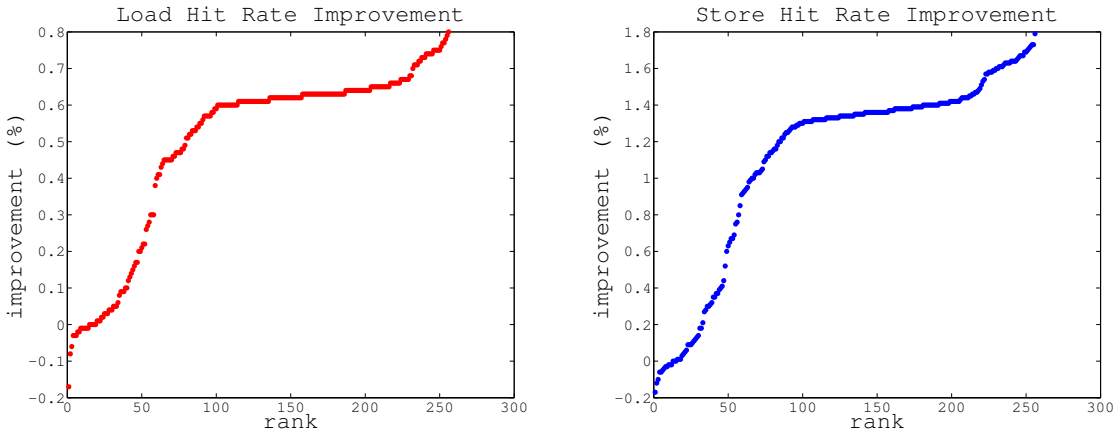



Figure 4: Load/Store Hit Rate when dimension=3 and looplevel=2

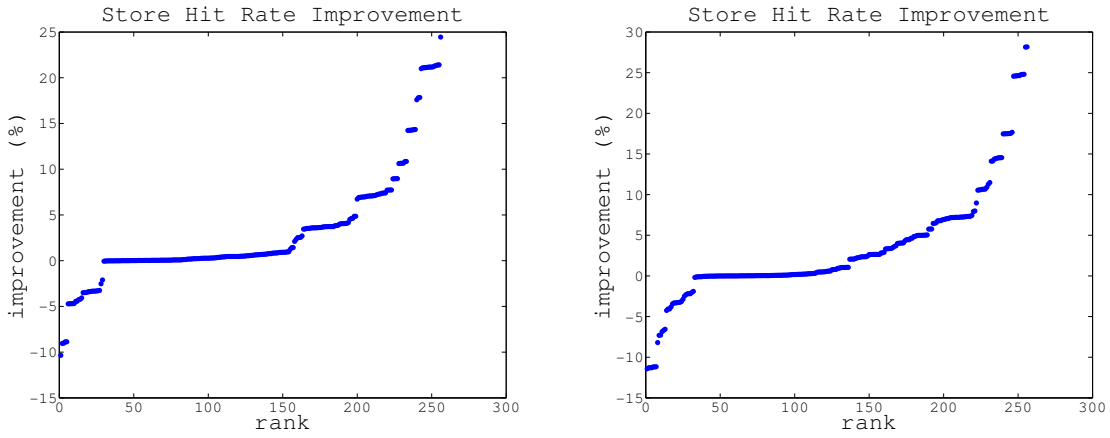



Figure 5: Store Hit Rate of two different accesses. Dimension=2 and looplevel=2. The two accesses in the left one has equal weights (both 10K iterations), and the two accesses in the right one has very different weights (10K and 100 iterations)

our algorithm can still strike a good balance even when compromising one access for another.

# 5 Surprises and Lessons Learned

1. When reorganizing the memory layout by solving linear equations, we find that the solutions do not necessarily make the transformed indices nonnegative. Currently we add an offset (can be computed by the size of the array) to the transformed array so that the transformed indices are guaranteed to be nonnegative.

2. When solving the linear equations for more than three variables, we find it is generally hard to get integer solutions. For two variables we can use the extended Euclidean algorithm, and for three variables we can further extend the algorithm by finding a particular solution of one variable first. However there does not exist general algorithms for more than three variables. Currently we are using integer programming method (searching solutions over a relatively small range), which is able to solve equations involving any number of variables. But it would be relatively slow when the searching range increases or when the number of variables is large.

# 6 Conclusion and Future Work

In this project we implemented memory layout reorganization in order to achieve array locality optimization. In contrast to tiling methods, we are able to optimize the code under the constraint that instruction orders should be fixed. We present a general

framework to reorganize memory layout, and propose a novel algorithm to find a better layout based on the access patterns we have analyzed. Moreover we extend our algorithm to handle the case when there are multiple access patterns in the code. Experiments on the benchmark show that our algorithm did improve cache performance.

For future work discussion, as now we only consider integer arrays in this project since it is easier to analyze and reorganize, in the future we may consider more complicated type of arrays, such as struct arrays. Moreover, we only did static analysis on the code, another possibility in the future can be dynamic analysis so that more complicated access patterns (indices not determined until execution) can be utilized for memory layout reorganization.

**Distribution of Total Credit**
The design of algorithms and the experiments are evenly distributed among group members, so we think the distribution of total credit should be 50%-50%.

# 7 References

[1] M. J. Wolfe. More iteration space tiling.
[2] A Compiler Framework for Optimization of Afne Loop Nests for GPGPUs, L Zuck, A Pnueli, B Goldberg, C Barrett, Y Fang
[3] Automatic Creation of Tile Size Selection Models, Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre Eichenberger and Kevin O'Brien.
[4] Parameterized Tiling Revisited, Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Tom Henretty, J Ramanujam, and P Sadayappan
[5] Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. S Ryoo, CI Rodrigues, SS Baghsorkhi
[6] Logical inference techniques for loop parallelization. Cosmin E. Oancea and Lawrence Rauchwerger.
[7] Speculative separation for privatization and reductions. Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August.