Fine-Granular Computation and Data Layout Reorganization for Improving Locality

Mahmut Kandemir
mtk2@psu.edu
Penn State University
State College, PA, USA

Xulong Tang
tax6@pitt.edu
University of Pittsburgh
Pittsburgh, PA, USA

Jagadish Kotra
Jagadish.Kotra@amd.com
AMD Research
Austin, TX, USA

Mustafa Karakoy
m.karakoy@yahoo.co.uk
TUBITAK-BILGEM
Turkey

ABSTRACT

While data locality and cache performance have been investigated in great depth by prior research (in the context of both high-end systems and embedded/mobile systems), one of the important characteristics of prior approaches is that they transform loop and/or data space (e.g., array layout) as a whole. Unfortunately, such coarse-grain approaches bring three critical issues. First, they implicitly assume that all parts of a given array would equally benefit from the identified data layout transformation. Second, they also assume that a given loop transformation would have the same locality impact on an entire data array. Third and more importantly, such coarse-grain approaches are local by their nature and difficult to achieve globally optimal executions. Motivated by these drawbacks of existing code and data space reorganization/optimization techniques, this paper proposes to determine multiple loop transformation matrices for each loop nest in the program and multiple data layout transformations for each array accessed by the program, in an attempt to exploit data locality at a finer granularity. It leverages bipartite graph matching and extends the proposed fine-granular integrated loop-layout strategy to a multicore setting as well. Our experimental results show that the proposed approach significantly improves the data locality and outperforms existing schemes – 9.1% average performance improvement in single-threaded executions and 11.5% average improvement in multi-threaded executions over the state-of-the-art.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**;
• **Software and its engineering → Compilers**.

KEYWORDS

Data locality, Data layout, Code optimization

ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3549386

ACM Reference Format:
Mahmut Kandemir, Xulong Tang, Jagadish Kotra, and Mustafa Karakoy. 2022. Fine-Granular Computation and Data Layout Reorganization for Improving Locality. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '22), October 30-November 3, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3508352.3549386

1 INTRODUCTION

Performance of many data-intensive application programs depends critically on their data access patterns and cache behavior. This is even more so in emerging multicore and manycore systems with multi-level cache hierarchies and high costs of last-level cache (LLC) misses. Observing these, prior researches have extensively investigated various ways of restructuring data access patterns to improve cache performance. Among proposed approaches to this problem are hardware-based prefetching [16], compiler-based access pattern restructuring [11, 38, 47], various data (memory) layout optimizations [4, 14, 15, 20, 35, 48], and more recent works on in-memory [2, 43] and near-data [17, 44] computing.

As far as pure compiler-based strategies are concerned, two different paradigms have been explored in the past to improve data locality (cache performance): *loop restructuring* and *data layout transformation*. In the former, the loop nest is transformed (restructured) by modifying the execution order of loop iterations. Among popular loop transformations that have been explored in the past are loop permutation, iteration space tiling, and loop skewing. In contrast, data layout transformations take an entirely different approach and instead of changing the execution order of loop iterations, they modify the memory layout of data. Examples include converting row-major layout to column-major layout and data space tiling. The papers published in compiler literature show reasonable improvements with loop transformations, data layout transformations, and techniques that combine both [23, 25–27, 38, 42, 50].

One of the important characteristics of these prior approaches is that they transform loop and/or data space *as a whole*. For example, an entire loop nest is transformed using a single loop transformation matrix (representing change of basis), and similarly, an entire data array is transformed via a layout transformation. Unfortunately, such coarse-grain approaches (when used in transforming loop nest and/or array data) bring three critical issues. First, these approaches implicitly assume that all parts of a given array would equally benefit from the identified data layout transformation, which may

not necessarily be true in practice. In fact, our analysis of different application programs clearly shows that different portions of an array can indeed prefer different layout transformations, primarily because they are accessed by different loop nests in different ways. For example, while one loop nest can access a given array in a row-major fashion, another loop can access the same array in a column-major fashion. Second, these approaches also implicitly assume that a given loop transformation would have the same locality impact on an entire array, which again may not necessarily be true. Third and maybe more importantly, such coarse-grain approaches are *local* by their nature. For example, each loop nest is transformed by considering the locality needs/behavior of that nest alone. However, in reality, applying a loop transformation imposes a certain access pattern over all array structures accessed in the loop nest, and that in turn requires a preferable loop transformation for other loop nests that access the same array structures. That is, in a sense, loop nests that access the same set of data arrays (or portions of arrays) need to be transformed with a global (program wide) view in mind, which is, unfortunately, missing in many prior works. Motivated by these drawbacks of existing techniques, this paper makes the following three main contributions:

- It proposes a novel data locality optimization strategy that employs both loop nest restructuring and data layout transformation in a *fine-granular fashion*. The proposed strategy is built upon maximum matching in bipartite graphs. It discusses why such a fine-granular strategy is expected to perform better than its coarse-granular counterparts.
- It extends the proposed fine-granular integrated loop-layout strategy to a multicore setting, and shows that bipartite graph matching can be used in a multicore context as well.
- It reports experimental evidence showing the effectiveness of the proposed strategy using 10 benchmark programs drawn from different sources. The reported results indicate around 9.1% average savings in execution cycles in single-threaded executions and 11.5% average savings in multi-threaded executions.

To our knowledge, this is the first work that considers integrated fine-grain loop and data layout optimizations for both single-threaded and multi-threaded executions. The rest of this paper is organized as follows. Section 2 gives the mathematical foundation upon which we build our proposed computation and data layout transformations. The technical details of our approach are presented in Section 3, and Section 4 briefly discusses its important aspects. A detailed experimental evaluation of the proposed approach is given in Section 5. The prior relevant work is discussed in Section 6 and the paper is concluded in Section 7 with a summary of our major observations/results and a brief discussion of the ongoing work.

2 BASICS

In this section, we go over the basics of compiler-based loop and data layout transformation theory, and introduce the basis of the linear algebraic formulation upon which our proposed computation and layout transformation strategies are built.

Our focus is on *affine loops* where loop bounds and array accesses (subscript functions) are affine functions of enclosing loop indices and loop-invariant variables and constants.[1] Note however that, if there is a non-affine array access, we just omit it (i.e., drop it from consideration as far as data locality optimization is concerned) and optimize the remaining references (i.e., we do not drop the entire loop nest from consideration). Iterations of a loop nest with n loops in our framework is represented by an iteration space and each point in this space is denoted by an iteration vector, namely, $\vec{I} = (i_1, i_2, \cdots, i_n)^T$. Each loop iterator, i_c (where $1 \leq c \leq n$) is bounded by two functions: $LB(i_c) = f(i_1, i_2, \cdots, i_{c-1})^T$ for lower bound and $UB(i_c) = (i_1, i_2, \cdots, i_{c-1})^T$ for upper bound. An array reference to an m-dimensional array in such a loop is represented by $\vec{d} = H\vec{I} + \vec{h}$, where H is an $m \times n$ matrix and $\vec{h}$ is an m-entry vector.

(a)
```
for(i1=2; i1≤N1; i1++)
  for(i2=i1; i2≤N2; i2++)
    ... X[i1+2][i1+i2-3] ...
```

(b)
```
for(i'1=2; i'1≤N2; i'1++)
  for(i'2=2; i'2≤min(i'1,N1,N2); i'2++)
    ... X[2i'2+2][i'1+i'2-3] ...
```

(c)
```
for(i'1=2; i'1≤N1; i'1++)
  for(i'2=i1; i'2≤N2; i'2++)
    ... X [i'1+i'2-3] [i'1+2] ...
```

Figure 1: Code example. a) Original code. b) After loop-transformation. c) After layout-transformation

Consider, as an example, the small code fragment in Figure 1(a). In this case, we have $\vec{I} = (i_1, i_2)^T$, $LB(i_1) = 2$, $BB(i_1) = N_1$, $LB(i_2) = i_1$, $UB(i_1) = N_2$, $H = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, and $\vec{h} = (2, -3)^T$.

In this framework, a loop transformation matrix T, which is essentially an $n \times n$ matrix for a loop nest with n loops, transforms a loop iteration vector $\vec{I}$ to $\vec{I'} = T\vec{I}$. Consequently, an original array access $H\vec{I} + \vec{h}$ is transformed under this loop transformation T to $HT^{-1}\vec{I'} + \vec{h}$. Figure 1(b) shows the loop-transformed version of the original code in Figure 1(a) with $T = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. We also want to mention that any loop transformation is constrained by intrinsic data dependences in the original code, and the new loop bounds after the transformation are determined using Fourier-Motzkin Elimination (FME [13]). Similarly, a data layout transformation (or simply layout transformation), for an m-dimensional array, can be represented by an $m \times m$ matrix M and the original array access, $H\vec{I} + \vec{h}$, is transformed under such a layout transformation, to $M(H\vec{I} + \vec{h}) = MH\vec{I} + M\vec{h}$. Clearly, a layout transformation does not affect loop bounds; however, array declarations need to be updated to reflect the new data layout. A layout-transformed version of the original code in Figure 1(a) with $M = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ is given in Figure 1(c).

3 PROBLEM FORMULATION

3.1 Definitions

Definition 1: *Computation Slab:* A computation slab (or, slab for short) is a rectilinear portion of a given iteration space. As an example, Figure 2(a) illustrates a rectilinear partitioning of the iteration space of a two-deep loop nest into 9 computation slabs.

Definition 2: *Data Tile:* A data tile represents a sub-portion of a multi-dimensional array (data space). Figure 2(b) depicts a

[1] Note that such loops are frequently found in scientific computing applications as well as embedded video/image processing applications.

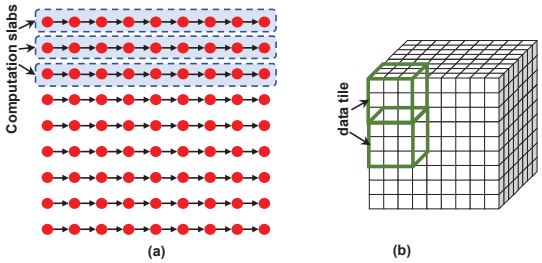

Figure 2: (a) Iteration space and computation slabs. (b) Data space the data tiles.

partitioning of the data space of a three-dimensional array into data tiles.

For a given loop nest N_i and an array X_j, the nth computation slab is denoted using $N_{i,n}$, and similarly, the mth data tile is denoted using $X_{j,m}$, with $1 \leq n \leq K_N$ and $1 \leq m \leq K_M$, where K_N and K_M represent, respectively, the total number of computation slabs and number of data tiles. A distinguishing characteristic of our proposed approach in this work is that each of $N_{i,n}$ (for a given X_i) can be transformed independently, and similarly, each $X_{j,m}$ (for a given X_j) can be transformed independently. It is important to emphasize that, for a given (N_i, X_j) pair, $N_{i,n}$ may or may not access $X_{j,m}$, depending on data access pattern exhibited by the loop nest. We note the following points:

- If $N_{i,n}$ does not access $X_{j,m}$, it should not have a say in how the memory layout of $X_{j,m}$ should be transformed.
- If both $N_{i,n}$ and $N_{i',n'}$ (where $i \neq i'$ or $n \neq n'$) access $X_{j,m}$, both of them should have a say on how the layout of $X_{j,m}$ should be transformed.
- If $N_{i,n}$ accesses both $X_{j,m}$ and $X_{j',m'}$ (where $j \neq j'$ or $m \neq m'$), the loop transformation chosen for $N_{i,n}$ should be compatible with the data layouts chosen for both $X_{j,m}$ and $X_{j',m'}$.

3.2 CDG: Computation-Data Graph

In this subsection, we introduce the main compiler data structure our approach employs. Note that this data structure, named CDG, is internal to the compiler and is user-transparent. A CDG is a bipartite graph $G(V_c, V_d, E, W)$, where each $v_q \in V_c$ represents a computation slab and $v_{q'} \in V_d$ represents a data tile. There is an edge $e \in E$ between $v_q \in V_c$ and $v_{q'} \in V_d$ if the computation slab represented by v_q (say, $N_{i,n}$) accesses the data tile represented by $v_{q'}$ (say, $X_{j,m}$), that is, there is *at least* one loop iteration in $N_{i,n}$ accessing a data element in $X_{j,m}$. For each edge $e = (v_q, v_{q'}) \in E$, we associate a weight $w \in W$ which captures the total number of references made by the iterations in v_q to the data elements in $v_{q'}$.[2]

Note that, CDG, in its most general form, represents the entire application program code. In the rest of our discussion, if there is no confusion, we will use the terms v_q and $N_{i,n}$ interchangeably, and similarly, we will use the terms $v_{q'}$ and $X_{j,m}$ interchangeably.

Given a CDG, our goal is to determine, for each $N_{i,n}$, a loop transformation $T_{i,n}$ and, for each $X_{j,m}$, a data layout transformation $M_{j,m}$ such that the overall data locality of the entire application program is improved. Figure 3(a) shows a sample CDG where $|V_c| = 9$, $|V_d| = 8$, and $|E| = 13$. Let us assume that all edges have the same weight (so, the weights are not shown explicitly).

[2]Clearly, this weight represents a compile-time estimation. Our approach also employs a limited symbolic analysis to compare weights symbolically (if they are not known at compile-time. Our approach can also use profile information when available.

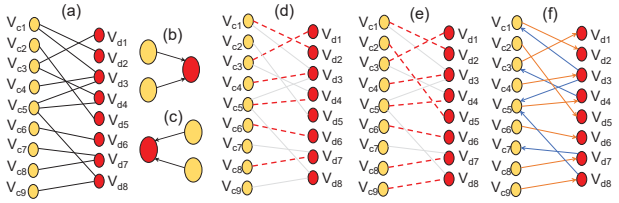

Figure 3: Example computation-data graph (CDG).

Let us discuss how this CDG presents constraints in assigning loop and data transformation matrices for the nodes in V_c and V_d. We can start, for example, with v_{c1} and determine a loop transformation matrix for it, say T_1. This loop transformation matrix in turn imposes a data access pattern on data tiles represented by nodes v_{d2} and v_{d3}. More specifically, we need to select data transformation matrices (M_2 and M_3) to improve (transform) that data access pattern of the data tiles represented by nodes v_{d2} and v_{d3} such that they become "cache friendly". Now, layout transformation M_3 in turn dictates loop transformations, T_4 and T_5, for computation slabs represented by nodes v_{c4} and v_{c5}, respectively. Then, T_5 dictates M_4 and M_8, and so on. Such a chained determination of loop and data transformation matrices is called a *flow* in the remainder of this paper. Thus, the goal of the compiler optimization in this work is to determine a flow on a given CDG that leads to the minimum number of cache misses, i.e., a flow that maximizes cache performance.

We now summarize the two main difficulties one might face when determining $T_{i,n}$ and $M_{j,m}$ transformation matrices.

- Figure 3(b) shows a scenario where two nodes in V_c (v_1 and v_2) access the same node (v_3) in V_d. The problem is that the two loop transformations selected for v_1 and v_2 may require *different* data transformations for v_3. Consequently, it may not be possible to find a data transformation for v_3 that satisfies the loop transformations preferred by v_1 and v_2. In the rest of this paper, this type of conflict is referred to as *Loop-to-Data Conflict* (LDC).
- Figure 3(c) depicts a scenario where two nodes in V_d, each with its own data layout transformations, demand *different* loop transformations for the same node in V_c. As a result, in this case, we may not be able to find a loop transformation that satisfies both the data transformations. This type of conflict in called *Data-to-Loop Conflict* (DLC) in the remainder of this paper.

Clearly, everything else being equal, we want to determine a flow on the CDG such that the number of LDCs and DLCs is *minimized*. Below, we present two strategies towards this goal – one targeting single-threaded execution and the other targeting multi-threaded execution.

3.3 Determining a Flow for Single-Threaded Execution

Our strategy for determining a flow is based on the concept of *Maximum Weighted Match in a Bipartite Graph*. Let us start by the following definitions:

Definition 3: *Matching in a Bipartite Graph:* Given a bipartite graph $G(V_c, V_d, E, W)$, a matching is a subset of the edges, say E', for which every node belongs to exactly one of the edges. Figure 3(d) shows a matching for the bipartite graph in Figure 3(a).

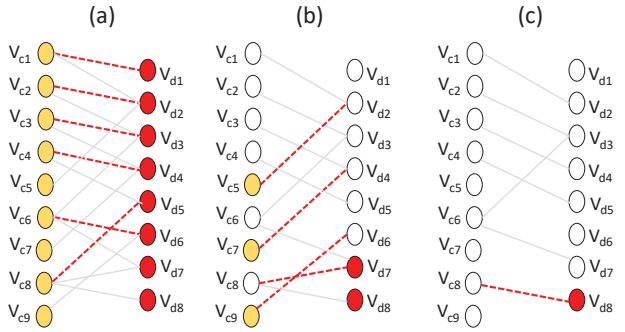

Figure 4: The residual graphs when MWM does not cover all node at initial step. The nodes being matched in each step are denoted using white color.

Definition 4: *Maximal Matching:* A matching (say E') of a bipartite graph $G(V_c, V_d, E, W)$ is said to be *maximal* if no other edges in $G(V_c, V_d, E, W)$ can be added to E'. That is, if another edge is added to E', E' would no longer be a matching. Note that, the matching depicted in Figure 3(d) is not maximal.

Definition 5: *Maximum Weight Matching (MWM):* Given a bipartite graph $G(V_c, V_d, E, W)$, an MWM is a *maximal matching* (say E'), when the total weights of the selected edges (the matched edges) is *maximum*. In other words, an MWM is the maximal matching with the total of the weights of the edges in E' is maximum. Figure 3(e) shows an MWM for the bipartite graph in Figure 3(a).

Let us now discuss why MWM is important in our problem of determining loop and data transformation matrices for an application program code represented by a bipartite graph $G(V_c, V_d, E, W)$. Note that, an MWM provides three nice properties as far as our data locality optimization problem is concerned. First, since MWM is a matching, performing code and data transformations (i.e., determining loop and data layout transformation matrices) based on it will avoid both LDCs and DLCs. That is, an MWM provides us with a *guideline* using which we can determine a *flow* (as defined earlier) on the target bipartite graph. Second, among all possible matchings in the target bipartite graph, MWM is the one that contains the most number of edges, i.e., it captures most data layout transformation matrix–loop transformation matrix interactions. This directly follows from the fact that MWM is maximal. Third, similarly, among all possible matchings in the target bipartite graph, MWM is the one that covers most weights, which means it takes into account the importance of different edges, i.e., the importance of the transformations of the two nodes that connect each important edge. For example, finding effective loop and data transformations for an edge with a large weight is more critical than finding effective loop and data transformations for an edge with a small weight. Note that, this third property directly follows from the fact that an MWM is a maximum matching. Overall, these three properties make MWM a good starting point for determining a flow on the target bipartite graph. In other words, our approach determines an *MWM-induced flow*, an example of which is shown in Figure 3(f).

Algorithm 1 gives the pseudo-code our compiler employs for determining a flow in a given target bipartite graph. Our algorithm starts by finding an MWM, denoted as E', on the bipartite graph[3]. It is known that, the problem of determining an MWM can be solved by converting it into a *flow network* and then using an algorithm to find the maximum flow on it[4]. In our implementation, we use the Ford-Fulkerson algorithm [18], to determine the flow on the generated flow network, though other algorithms such as those on augmenting paths [22]. Since the details of how to convert a matching problem into a flow problem are well known, we do not discuss it here in detail.

Clearly, *not* every MWM will cover *all* nodes (a simple example is shown in Figure 4(a)), yet we need to determine a loop transformation for each node in V_c and a data layout transformation for each node in V_d. So, we then remove all edges in E' as well as the nodes to which they are connected from the bipartite graph and obtain what can be termed as a *Residual Graph*. We next find an MWM on this residual graph and determine (the so-far-undetermined) loop and data transformations. It is to be noted however that, in determining a loop and data transformation in the MWM of this residual graph, we may run into conflicts, an example of which is illustrated in Figure 4(b). After this step, the newly-determined MWM (say E'') is removed from the residual graph, obtaining a new residual graph on which we find a new MWM, and so on. This cycle of **(1)** *find an MWM on the current residual graph* ⇒ **(2)** determine loop and data layout transformations for the nodes in MWM ⇒ **(3)** *remove MWM from the current residual graph to obtain a new residual graph* continues until we determine transformation (loop/data) matrices for *all* nodes in the original (input) bipartite graph.

Consider now Figure 4(c), which illustrates how Algorithm 1 operates on the sample bipartite graph depicted in Figure 4(a). In this case, the algorithm generates 3 residual graphs to determine the required loop and data transformation matrices.

3.3.1 Mechanics of Transformation. Let us now go over how we actually determine the loop and data transformations for a given MWM (the process is repeated for each and every MWM). Basically, in a given MWM, we process the edges one by one starting with the edge with the largest weight. Let us consider, without loss of generality, an edge $e_{ci,dj}$, i.e., the edge between nodes V_{ci} and V_{dj}. There are four cases to consider at this point:

- *Case-1:* We have already a determined loop transformation (from the previous step) for V_{ci} but have not determined yet a data transformation for V_{dj}. Note that, based on our discussion in Section 2, an array reference such as $H\vec{I} + \vec{h}$ – corresponding to the edge between V_{ci} and V_{dj} – is transformed under loop transformation T and data transformation M to $MHT^{-1}\vec{I'} + M\vec{h}$. If T has already been determined, we just determine M in this expression such that the resulting access pattern in $MHT^{-1}\vec{I'}+M\vec{h}$ becomes stride-1 (sequential).
- *Case-2:* This is the opposite of the previous scenario. In this case, we already have an M determined for V_{dj} and but do not yet have a T determined for V_{ci}. So, we determine T in $MHT^{-1}\vec{I'} + M\vec{h}$ such that the resulting access is either stride-1 or is independent of the innermost loop iterator (after the transformation).[5].

[3]Note that, an MWM can also be seen as a graph (not just as an edge set) by including all edges in E' as well as the nodes that those edges are attached to.

[4]Note that the term "flow" here is clearly different from the term "flow" we have been using so far – the latter denotes a solution that gives us the loop and data layout transformation matrices.

[5]It should be observed that, while a loop transformation can improve both temporal and spatial locality, a data transformation can improve only spatial locality [25]

Algorithm 1 MWM-guided computation and data layout transformations.

INPUT: a bipartite graph $G(V_c, V_d, E, W)$, where each $v_q \in V_c$ represents a computation slab and $v_{q'} \in V_d$ represents a data tile
OUTPUT: (computation slab, data tile) pairs that satisfies MWM
1: Residual graph is initialized to G
2: $R(V_c, V_d, E, W) \leftarrow G(V_c, V_d, E, W)$
3: output $\leftarrow \emptyset$
4: **while** V_c is not $\emptyset$ or V_d is not $\emptyset$ **do**
5: $list\_(V_c^{'}, V_d^{'}) \leftarrow$ Ford-Fulkerson$G$
6: $V_c \leftarrow V_c$ - $V_c^{'}$ and $V_d \leftarrow V_d$ - $V_d^{'}$
7: **if** check_conflict(output, $list\_(V_c^{'}, V_d^{'})$) **then**
8: merge $(V_c^{'}, V_d^{'})$ with (V_c, V_d), where $V_d^{'} = V_d$ or $V_c^{'} = V_c$
9: remove $(V_c^{'}, V_d^{'})$ from $list\_(V_c^{'}, V_d^{'})$
10: **end if**
11: output $\leftarrow$ output $\cup$ $list\_(V_c^{'}, V_d^{'})$
12: **end while**
13: //identify loop transformation and data layout transformation on each computation slab and data tile in $output$

Algorithm 2 High-level view of the extension to handle multi-threaded applications.

INPUT: a bipartite graph $G(V_c, V_d, E, W)$, where each $v_q \in V_c$ represents a computation slab and $v_{q'} \in V_d$ represents a data tile; number of partitions/cores (p)
OUTPUT: partitioning of computation slabs across cores, and for each core, (computation slab, data tile) pairs that satisfies MWM
1: Partition V_c into p partitions, $V_{c1}, V_{c2}, \cdots, V_{cp}$, such that
2: $\sum W_{ij}$ is minimized, where (a) $C(E_{ij}) \in V_{ci}$ and $D(E_{ij}) \in V_{dj}$, where C and D give, respectively, the sets to which the left end and right end of E_{ij}, and (b) $|V_{c1}| \approx |V_{c2}| \approx \cdots \approx |V_{cp}| \approx |V_c|/p$.
3: **for do**k from 1 to p
4: initialize any node in V_{ck} and V_{dk} from the previous partition
5: Call Algorithm 1 ($G(V_{ck}, V_{dk}, E_k, W_k)$. where V_{dk} is the set of data tiles accessed by V_{ck}; E_k is the set of edges that connect V_{ck} and V_{dk}; and W_k is the weights attached to the edges in E_k
6: **end for**

- *Case-3:* No T and no M have been determined so far (for V_{ci} and V_{dj}, respectively). In this case, we set either T or M to the identity matrix and determine the other.
- *Case-4:* We have already have a T and M determined for V_{ci} and V_{dj}, respectively. In this case, we do not do anything else (we have no scope for optimizing this edge; the prior optimizations impose a layout as well as a loop transformation).

3.4 Determining Flows for Multi-Threaded Execution

Note that Algorithm 1 described above operates on a single-threaded code. In this section, we discuss another compiler algorithm, Algorithm 2, that targets multi-threaded application programs. Like Algorithm 1, Algorithm 2 also works on a bipartite graph-based representation of the program data access pattern; however, it takes into account the on-chip cache hierarchy of the target architecture in assigning computations (actually computation slabs) to cores. Below, we discuss our approach in two cases – first without dependencies across computation slabs and then with dependencies across computation slabs.

The core idea behind our approach to determining loop and data transformations for computation slabs and data tiles, respectively, in the context of multi-threaded execution is to *divide* the target bipartite graph into multiple partitions. More specifically, we divide V_c of $G(V_c, V_d, E, W)$ into p partitions where p is the number cores. The main goal of this partitioning is to *minimize* the number of edges that cross partition boundaries. In mathematical terms, suppose that we have p divisions of V_c, namely $V_{c1}, V_{c2}, \cdots, V_{cp}$ such that $V_{c1} \cup V_{c2} \cup \cdots \cup V_{cp} = V_c$ and $V_{ci} \cap V_{cj} = \emptyset$ for any $i \neq j$. For a given V_{ci}, we define V_{di} as the set of data tiles accessed by the computation slabs. Clearly, in general, we have $V_{di} \cap V_{dj} \neq \emptyset$.

Now, we want to determine a partitioning such that

$$\sum W_{ij},$$

is minimized, where (a) $C(E_{ij}) \in V_{ci}$ and $D(E_{ij}) \in V_{dj}$, where C and D give, respectively, the sets to which the left end and right end of E_{ij}, and (b) $|V_{c1}| \approx |V_{c2}| \approx \cdots \approx |V_{cp}| \approx |V_c|/p$. While the first condition ensures that the edge in question crosses partition boundaries, the second condition ensures load balance. If all the weights are the same, then it means we are looking for a partitioning that minimizes

$$\sum_{i \neq j} |V_{di} \cap V_{dj}|,$$

i.e., one that minimizes the data tile sharing across computation slabs.

Two different partitionings of the sample CDG in Figure 5(a) are illustrated in Figures 5(b) and (c). The one in (c), which is the minimum weight partitioning in this case, is much better than the one in (b).

Note that, this particular partitioning we are proposing is oriented towards minimizing data tile sharing among computation slabs. After this partitioning has been performed, we apply Algorithm 1 to each partition (as in captured by the loop in Algorithm 2). In doing so, the LDC and DLC types of conflicts are resolved, in the order they are encountered, depending on the processing order of partitions. For example, we first process a partition and determine the loop and data layout transformation matrices for the computation slabs and data tiles in that partition. Then, when we go to the next partition, if this second partition shares some data tiles with the previous one, the already-determined data layout matrices (in processing the first partition) are used to initiate the flow determination process for this second partition. In this way, each time the algorithm comes to process a partition, it first performs the *initializations* dictated by the previously processed partitions. Algorithm 2 gives the pseudo-code for our fine-granular locality optimization algorithm targeting multi-threaded application programs.

4 DISCUSSION

In this section, we go over a couple of important aspects of our proposed approach. First, our approach is quite general and is applicable to any loop nest that is analyzable by the compiler. As stated earlier, if there are unanalyzable references in the loop nest being optimized, our approach simply omits them during locality optimization[6], and focuses on the remaining references. Second, while our approach increases both the compilation time and code size a bit (as will be reported in the next section), considering the significant execution time improvements it brings, we believe such increases are acceptable in many execution environments. Third, to determine the edge weights, our approach uses extensive compiler

[6]Note, however, that such references are treated conservatively when computing data and control dependencies.

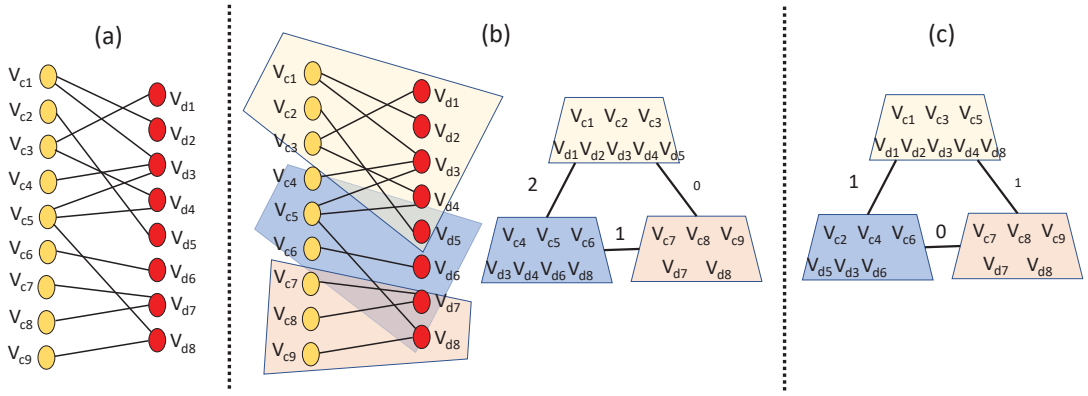

Figure 5: Two different potential partitionings of a given CDG.

analysis and symbolic analysis (to compare compile-time unknown variables). We want to emphasize that our approach is built in a modular fashion and can easily accommodate a more accurate weight estimator should one be available in the future. Fourth, since our approach is mainly built upon carrying over the constraints (loop transformations and layout transformations) determined in the past to the future (it is a greedy heuristic), it may, in some cases, end up in a situation where it is unable to transform, for example, a computation slab (because of conflicting layout constraints) or a data tile (because of conflicting computation transformations). In such cases, it does not perform any transformation for the conflicting case, i.e., it uses identity transformation. Fifth, although our compiler-based approach is applicable to any application program, in this study, we present results with 10 programs chosen from various sources. In selecting these benchmarks, we paid special attention to ensuring that (i) for each benchmark, both single-threaded and multi-threaded versions are available and (ii) they are representative in terms of variety of data access and data reuse patterns (e.g., sequential vs. stride and regular vs. irregular accesses). Finally, since our approach is built in a modular fashion in LLVM, it can be easily integrated with existing (and potentially future) optimizations in LLVM. A detailed study of such inter-optimization interactions, however, is postponed to a future study.

5 EXPERIMENTAL EVALUATION

5.1 Setup

In our evaluations, we used an Intel Broadwell processor core i5 clocked at 2.7GHz. Note that this processor is frequently employed in mobile and embedded settings. Its last-level cache (L3) has a capacity of 6MB, and it has 8 cores, each running 1 thread.

We implemented our approach in the LLVM 9.0.0 [32] compilation tool-set and tested its effectiveness using 10 applications using their single-threaded and also multi-threaded versions. In the case of single-threaded applications, our approach increased the average compilation time (over the baseline defined below) by about 25% and, in the case of multi-threaded applications, the average increase in compilation time was about 36%. Also, the average code size increases were 22% and 28% in the cases of single-threaded and multi-threaded application programs, respectively.

Below, we present the results from single-threaded programs and multi-threaded programs separately. Table 1 lists our application programs, their dataset sizes, the last-level cache misses of their original versions, and their execution times (the multi-threaded results are collected using 8 cores [1 thread/core]). *blob* and *canny* are two locally-maintained application programs, based on algorithms discussed in [8] and [36], respectively. The first one is an implementation of the Canny edge detection algorithm, whereas the second one is a blob detection program, which tries to detect regions in an image that differ in various properties, compared to surrounding regions. The remaining eight application programs are from three different benchmark suites [5, 7, 51]. We selected these applications such that our suite has diversity of (i) dataset sizes, (ii) data access/reuse patterns, and (iii) last-level cache performance.

Table 1: List of applications.

Application	Dataset Size (MB)	LLC Misses (%)	Exec Time (sec) [single-core]	Exec Time (sec) [multi-core]
blob	366.1	28.1	4.21	0.68
canny	1,021.40	37.2	7.87	1.29
facesim [7]	582.3	19.3	3.91	0.62
x264 [7]	596.4	22.6	2.28	0.43
radix [51]	1,193.30	38.7	6.11	1.24
raytrace [51]	338.1	18.9	2.81	0.77
volrend [51]	785.2	31.5	5.83	1.12
swim [5]	672.4	24.6	5.24	0.96
imagick [5]	888.2	27.3	4.8	0.82
kdtree [5]	1,202.40	36.6	9.17	1.66

Note that, the results presented in Table 1 are collected with the Intel machine described above, and to put pressure on cache hierarchy, we increased the dataset sizes of the application programs in our experimental suite. As can be noted from this table, as far as cache behavior is concerned, our applications exhibit great variety, with LLC misses ranging from 18.9% to 38.7%. Note also that, all these values shown in the table are collected using the original applications with the conventional data locality optimization. More specifically, these applications have been optimized (even in their default versions in our setup) with all major data locality optimizations such as loop permutation and loop tiling[7], and all low-level parallelism optimizations such as SIMD/vectorization. The reported performance improvements presented below are *normalized* with respect to these (already locality-optimized) *baseline* versions.

5.2 Main Results

5.2.1 Single-Threaded Execution Results. The percentage reductions in last-level cache (L3) misses and execution times (both with

[7]For each loop nest in each benchmark, we experimented with different tile sizes and used the one that performed the best among all.

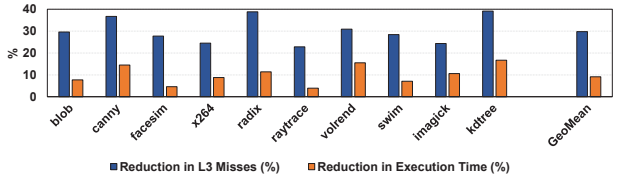

Figure 6: The percentage reductions in last-level cache (L3) misses and execution times with single-threaded executions.

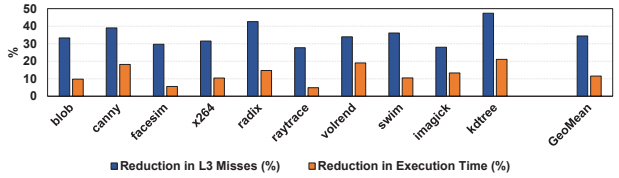

Figure 7: The percentage reductions in last-level cache (L3) misses and execution times with multi-threaded executions.

respect to the original application executions) are plotted in Figure 6, in the case of single-threaded executions. It can be observed from these results that our proposed compiler approach brings cache miss reductions ranging between 22.8% and 39.1% (averaging 29.6%) and total execution time reductions ranging between 3.9% and 16.7% (averaging on 9.1%). We believe that a 9.1% average execution time improvement is significant, considering the fact that, as explained before, the original codes had already heavily been *optimized for data locality*.

5.2.2 Multi-Threaded Execution Results. The results with the multi-threaded executions are plotted in Figure 7. It can be observed that, on average, our proposed compiler support improves LLC misses and execution times by 34.4% and 11.5%, respectively. It is to be noted that, compared to the single-threaded executions, these improvements are higher, primarily because the original programs exhibit poorer locality in the multi-threaded case due to the loss of some locality as a result of partitioning iteration spaces across multiple cores (leading to discontinuity in stride-1 accesses to data).

5.2.3 Application and Optimization Statistics. We next give some statistics explaining the operation of the proposed compiler support. The second, third and fourth columns of Table 2 give the bipartite graph parameters for each of our applications in the default setting used: the number of computation slab nodes ($|V_c|$), the number of data tile nodes ($|V_d|$), and the number of edges ($|E|$). The next two columns of the same table give the number of the two types of conflicts discussed earlier: LDC and DLC. The numbers listed in these columns give the absolute numbers of LDC and DLC our compiler experienced, and the percentage values within parentheses indicate the fractions of them that our compiler has successfully resolved *without* compromising any locality – meaning that, in case of LDC, the two loop transformation matrices were found to be the same and in case of DLC, the two data layout transformation matrices we found to be the same.

One can make two main observations from these results. First, the total number of conflicts (|LDC|+|DLC|) is quite small for each benchmark, considering the bipartite graph sizes. Second and more importantly, in an overwhelming majority of the cases where a conflict occurs, the compiler was able to find a transformation (loop or data) that resolves the conflict. More specifically, on average,

Table 2: Bipartite graph parameters for each application.

Application	|Vc|	|Vd|	|E|	|LDC|	|DLC|
blob	4,124	3,428	4,099,751	266484 (72.2%)	317321 (69%)
canny	8,802	6,636	11,602,518	638138 (87.3%)	241479 (91.4%)
facesim	5,556	3,890	7,861,264	338034 (71.2%)	221972 (75.7%)
x264	6,074	4,844	9,120,961	592862 (59.5%)	438536 (53.9%)
radix	8,488	7,006	12,082,724	604136 (89.4%)	739269 (93.2%)
raytrace	3,996	3,276	2,356,361	65978 (72.6%)	84289 (68.2%)
volrend	6,896	5,194	9,968,951	328975 (88.1%)	278383 (87.5%)
swim	4,924	4,138	4,908,898	88360 (64.4%)	95743 (68.2%)
imagick	7,338	5,822	10,253,241	461396 (55.2%)	429991 (58.6%)
kdtree	8,988	6,964	8,388,865	293610 (90.5%)	362085 (96.2%)

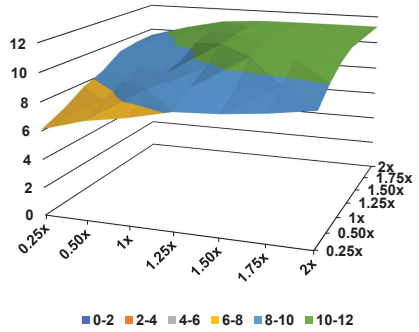

Figure 8: Results of single-core execution.

about 75% (resp. 76.2%) of the LDC type (resp. DLC type) of conflicts got resolved without sacrificing any data locality. In the remaining cases, the transformation matrices involved in the conflict were not the same, and our compiler selected one of them randomly, resulting in some loss in data locality.

5.3 Sensitivity Analysis

The main sensitivity analysis we perform is to change the numbers of computation slabs and data tiles. Recall that the default numbers used in our experiments so far are given in Table 2. Figures 8 and 9 plot the percentage improvements in execution time for single-core and multi-core executions, respectively, when the numbers of computation slabs and data tiles are modified. Note that these graphs plot the geometric mean values across all applications tested. In these plots, $1x$ refers to the default numbers, whereas nx refers to the case where the computation slab or data tile count is set of n times of the original count. In both these graphs, the x-axis and y-axis capture, respectively, the normalized number of computation slabs and the number of data tiles. The z-axis, on the other hand, captures the percentage improvement in execution time over the original execution. The main observation from these results is that, as the number of slabs or tiles is increased, our savings also increase. However, the savings saturate with very large slab/tile counts, primarily because most of the transformation matrices representing increasingly small parts of iteration and data spaces become very similar (as a result, any further partitioning does not help).

5.4 Comparison against Related Work

As has been mentioned above, the baseline codes that we used in our experimentation have already been aggressively optimized for data locality. Consequently, we believe that the average savings achieved by our proposed approach (9.1% in single-threaded execution and 11.5% in multi-threaded execution) are significant. We also compared our approach against an alternate approach that

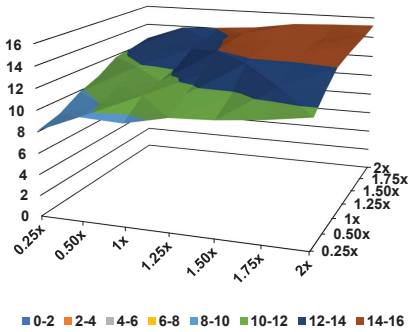

Figure 9: Results of multi-core execution.

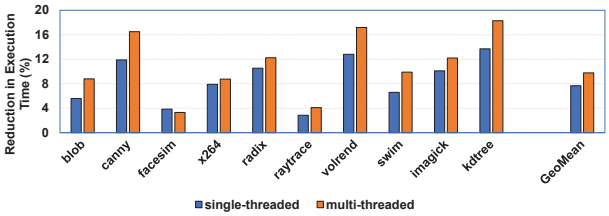

Figure 10: The percentage execution time reduction brought by our approach over the prior work.

uses *both* loop and data transformations. This approach has been implemented based on the algorithm presented in [23], but additionally, it also uses iteration space tiling (again, the tile sizes have been determined via experimentation) for locality optimization. The results plotted in Figure 10 show the execution time improvements for single-threaded and multi-threaded versions, brought by our approach over the approach in [23]. It can be observed that, the average improvements (GeoMean) brought by our proposed approach are 7.7% and 9.8% for single-threaded and multi-threaded applications, respectively. These results emphasize, in our opinion, the importance of fine-grained loop and data layout optimization, as opposed to determining only a single computation transformation matrix for each loop nest and a single layout transformation matrix for each data array.

6 RELATED WORK

Data locality optimizations. The compiler literature has substantial prior works focusing on optimization techniques that target improve the cache performance. Representative works can be broadly divided into two bodies: code (access pattern) restructuring [9, 11, 24, 28–31, 33, 34, 38, 45–47, 49] and/or data layout reorganization [12, 37, 40]. A data cache organization called dual data cache is proposed by Gonzalez et al. to manage spatial and temporal locality independently [19]. Wolf and Lam [49] proposed unimodular transformations and tiling to leverage temporal and spatial reuse for data locality. Chatterjee et al. [10] explored nonlinear array layout functions as a means of improving the locality of reference. Ding et al. [15] presented compiler techniques that reduce the number of on-chip hops for an off-chip request. Such an approach reduces the queuing delays for the off-chip requests, thereby increasing the performance. Piccoli et al. [41] proposed compiler and runtime support for dynamically memory page migration to improve data locality on NUMA architectures. Our work is different from these prior efforts as we focus on a finer granularity instead of treating the entire loop body as a whole. As a result, our approach is able to leverage the opportunities that different portions of an array prefer different layouts. Furthermore, our approach considers the global optimization of array layout, which is different from prior works optimizing loop body locally.

Near data computing. Another widely adopted approach to reducing the data access latency is near-data computing (NDC). Actually, the concept of NDC is not new and can be traced back to 1970s. Recently, due to the florescent of new memory technologies such as 3D-stacked memory and non-volatile memory, NDC gain a momentum [1–3, 6, 21, 39, 53]. Xu et al. [52] proposed processing-in-memory (PIM) architectures for deep learning algorithms to execute simple operations near the operands. Zhang et al. [53] proposed a 2.5D based PIM architecture named TOP-PIM. They showed significant energy savings and performance improvements when executing certain workloads closer to the memory. Compared to these prior efforts, our proposed approach focuses on fine-granular data locality optimization and is orthogonal to most near-data computing optimizations. If needed, our approach can be easily combined with near-data computing optimizations to further improve the data access performance, hence improving the overall application performance.

7 CONCLUDING AND FUTURE WORK

While prior research works have substantially optimized data locality and cache performance in the context of both high-end systems and embedded/mobile systems, to our knowledge, all existing works that consider both computation restructuring (loop transformations) and data restructuring (layout optimizations) determine one loop transformation for each loop nest and one data layout transformation for each multi-dimensional data array. The main abstraction our compiler employs is both loop nest restructuring and data layout transformation in a fine-granular manner. The proposed compiler leverages the maximum matching in bipartite graphs to automatically determine the granularity for both single-threaded and multi-threaded applications. The experimental results are collected using both single-threaded and multi-threaded versions of 10 benchmark programs. The results indicate that the proposed compiler achieves 9.1% average time reduction in single-threaded executions and 11.5% average time reduction in multi-threaded executions.

Our ongoing work includes (i) exploring interactions between our proposal and other data locality optimizations in more detail; (ii) studying interactions between different code parallelization strategies and our approach; and (iii) extending our approach to take advantage of emerging hardware accelerators in state-of-the-art multicore/manycore systems including GPUs.

ACKNOWLEDGEMENT

The authors would also like to thank the anonymous reviewers for their constructive feedback and suggestions. This work is supported in part by NSF grants #2119236, #2211018, #1763681, #2028929, #2011146, #2154973, and #1931531, as well as a startup funding from the University of Pittsburgh. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *ISCA*.

[2] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *ISCA*.

[3] Berkin Akin, Franz Franchetti, and James C. Hoe. 2015. Data Reorganization in Memory Using 3D-stacked DRAM. In *ISCA*.

[4] Jennifer M. Anderson and Monica S. Lam. 1993. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *PLDI*.

[5] Vishal Aslot, Max J. Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. 2001. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *WOMPAT*.

[6] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-data processing: Insights from a MICRO-46 Workshop. *Micro, IEEE* (2014).

[7] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph. D. Dissertation. Princeton University.

[8] John Canny. 1986. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence* 6 (1986), 679–698.

[9] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. 1994. Compiler Optimizations for Improving Data Locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[10] Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck Shyam Mundhra, and Mithuna Thottethodi. 1999. Nonlinear array layouts for hierarchical memory systems. In *Proc. of ICS*.

[11] G. Chen and M. Kandemir. 2005. Code Restructuring for Improving Cache Performance of MPSoCs. In *DAC*.

[12] Michal Cierniak and Wei Li. 1995. Unifying Data and Control Transformations for Distributed Shared-memory Machines. In *PLDI*.

[13] G.B. Dantzig and B.C. Eaves. 1973. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial T heory* 14, A (1973), 288–297.

[14] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *ASPLOS*.

[15] Wei Ding, Xulong Tang, Mahmut Kandemir, Yuanrui Zhang, and Emre Kultursay. 2015. Optimizing Off-chip Accesses in Multicores. In *PLDI*.

[16] Hongyu Fang, Sai Santosh Dayapule, Fan Yao, Miloš Doroslovački, and Guru Venkataramani. 2018. Prefetch-guard: Leveraging hardware prefetches to defend against cache timing channels. In *HOST*.

[17] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. 2015. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *HPCA*.

[18] Lester Randolph Ford Jr and Delbert Ray Fulkerson. 2015. *Flows in networks*. Princeton university press.

[19] Antonio González, Carlos Aliagas, and Mateo Valero. 2014. A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality. In *ICS*.

[20] Mary H. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. 1995. Detecting Coarse-grain Parallelism Using an Interprocedural Parallelizing Compiler. In *Supercomputing*.

[21] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *ISCA*.

[22] Roy Jonker and Anton Volgenant. 1987. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing* 38, 4 (1987), 325–340.

[23] Mahmut Kandemir, A Choudhary, J Ramanujam, and Prithviraj Banerjee. 1998. Improving locality using loop and data transformations in an integrated framework. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 285–296.

[24] Mahmut Kandemir, Alok Choudhary, J Ramanujam, and Prith Banerjee. 1999. A matrix-based approach to global locality optimization. In *Journal of Parallel and Distributed Computing*.

[25] Mahmut Kandemir, Alok Choudhary, Nagaraj Shenoy, Prithviraj Banerjee, and J Ramenujarn. 1999. A linear algebra framework for automatic determination of optimal data layouts. *IEEE Transactions on Parallel and Distributed Systems* 10, 2 (1999), 115–135.

[26] Mahmut Kandemir, Ismail Kadayif, and Ugur Sezer. 2001. Exploiting scratch-pad memory using presburger formulas. In *Proceedings of the 14th international symposium on Systems synthesis*. 7–12.

[27] Mahmut Kandemir, J Ramanujam, and Alok Choudhary. 1997. A compiler algorithm for optimizing locality in loop nests. In *Proceedings of the 11th international conference on Supercomputing*. 269–276.

[28] Mahmut Kandemir, J. Ramanujam, Alok Choudhary, and Prithviraj Banerjee. 2001. A layout-conscious iteration space transformation technique. In *IEEE Transactions on Computers*.

[29] Orhan Kislal, Jagadish Kotra, Xulong Tang, Mahmut Taylan Kandemir, and Myoungsoo Jung. 2018. Enhancing Computation-to-core Assignment with Physical Location Information. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[30] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. 1997. Data-centric Multi-level Blocking. In *PLDI*.

[31] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[32] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[33] Shun-Tak Leung and John Zahorjan. 1995. *Optimizing data locality by array restructuring*. Department of Computer Science and Engineering, University of Washington, Seattle, WA.

[34] Wei Li. 1994. *Compiling for NUMA parallel machines*. Technical Report. Cornell University.

[35] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. 1999. An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. In *ICS*.

[36] Tony Lindeberg. 2013. Scale selection properties of generalized scale-space interest point detectors. *Journal of Mathematical Imaging and vision* 46, 2 (2013), 177–210.

[37] Qingda Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Yongjian Chen, Haibo Lin, and Tin-Fook Ngai. 2009. Data Layout Transformation for Enhancing Data Locality on NUCA Chip Multiprocessors. In *PACT*.

[38] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. 1996. Improving Data Locality with Loop Transformations. *ACM Trans. Program. Lang. Syst.* 18, 4 (1996).

[39] R Nair, SF Antao, C Bertolli, P Bose, JR Brunheroto, T Chen, C-Y Cher, CHA Costa, C Evangelinos, and BM Fleischer. 2015. Active Memory Cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development* (2015).

[40] M.F.P. O'Boyle and P.M.W. Knijnenburg. 2002. Integrating Loop and Data Transformations for Global Optimization. *J. Parallel Distribute Computer* (2002).

[41] Guilherme Piccoli, Henrique N. Santos, Raphael E. Rodrigues, Christiane Pousa, Edson Borin, and Fernando M. Quintão Pereira. 2014. Compiler Support for Selective Page Migration in NUMA Architectures. In *PACT*.

[42] Samuel Rogers and Hamed Tabkhi. 2018. Locality Aware Memory Assignment and Tiling. In *Proceedings of the 55th Annual Design Automation Conference*.

[43] William Simon, Juan Galicia, Alexandre Levisse, Marina Zapater, and David Atienza. 2019. A fast, reliable and wide-voltage-range in-memory computing architecture. In *DAC*.

[44] Gagandeep Singh, Juan Gómez-Luna, Giovanni Mariani, Geraldo F Oliveira, Stefano Corda, Sander Stuijk, Onur Mutlu, and Henk Corporaal. 2019. NAPEL: Near-memory computing application performance prediction via ensemble learning. In *DAC*.

[45] Yonghong Song and Zhiyuan Li. 1999. New Tiling Techniques to Improve Cache Temporal Locality. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI)*.

[46] Xulong Tang, Mahmut Taylan Kandemir, Mustafa Karakoy, and Meena Arunachalam. 2019. Co-Optimizing Memory-Level Parallelism and Cache-Level Parallelism. In *Proceedings of the 40th annual ACM SIGPLAN conference on Programming Language Design and Implementation*.

[47] Kelefouras Vasilios, Keramidas Georgios, and Voros Nikolaos. 2018. Combining Software Cache Partitioning and Loop Tiling for Effective Shared Cache Management. *ACM Transactions on Embedded Computing System.* (2018).

[48] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *ASPLOS*.

[49] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. In *PLDI*.

[50] Wayne Wolf and Mahmut Kandemir. 2003. Memory system optimization of embedded software. *Proc. IEEE* 91, 1 (2003), 165–182.

[51] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*.

[52] Lifan Xu, Dong Ping Zhang, and Nuwan Jayasena. 2015. Scaling Deep Learning on Multiple In-Memory Processors. In *Proceedings of the 3rd Workshop on Near-Data Processing*.

[53] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *HPDC*.