2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), and 2015 IEEE 12th International Conf on Embedded Software and Systems (ICESS)

Gregarious Data Re-structuring in a Many Core Architecture

Sunil Shrestha
University of Delaware
Email: sunil@udel.edu

Joseph B. Manzano
Pacific Northwest National Lab
Email: joseph.manzano@pnnl.gov

Andres Marquez
Pacific Northwest National Lab
Email: andres.marquez@pnnl.gov

Stephane Zuckerman
University of Delaware
Email: szuckerm@udel.edu

Shuaiwen Leon Song
Pacific Northwest National Lab
Email: shuaiwen.song@pnnl.gov

Guang R. Gao
University of Delaware
Email: ggao@udel.edu

***Abstract*— As new massively multithreaded many-core architectural designs continue to evolve, the challenge of finding schedules that exploit concurrency, reuse and locality remains. Classically, data transformations were made with a limited view of both the memory hierarchy and the parallelism available to the machine. However, as the multithreaded designs become more complex, the machine resources tend to get "grouped" or shared in more sophisticated arrangements (e.g. distributed Level 2 caches serving collectively as a Level 3 cache or a large number of simultaneous multithreading). These new configurations present new optimization opportunities that the software toolchains might not be aware and, therefore, miss altogether.**

In this paper, we have developed a new methodology that takes in consideration the access patterns from a single parallel actor (e.g. a thread), as well as, the access patterns of "grouped" parallel actors that share a resource (e.g. a distributed Level 3 cache). We start with a hierarchical tile code for our target machine and apply a series of transformations at the tile level to improve data residence in a given memory hierarchy level. The contribution of this paper includes (a) collaborative data restructuring for group reuse and (b) low overhead transformation technique to improve access pattern and bring closely connected data elements together. Preliminary results in a many core architecture, Tilera TileGX, shows promising improvements over optimized OpenMP code (up to 31% increase in GFLOPS) and over our own previous work on fine grained runtimes (up to 16%) for selected kernels.

I. Introduction

As the parallelism of current and future machines continues to increase, the application and system designers are confronted with a new set of problems on how to exploit the computational power of these designs. As more and more of these designs become available, one of the major factors of inefficiencies is the access (and exploitation) of the memory and its associated hierarchy. In efforts to remedy the memory latency issue, compiler techniques such as tiling [1], [2], [3] and prefetching [4] have been widely used. Also, communication-avoiding algorithms [5] and cache-oblivious algorithms [6] have shown promises in reducing the number of high latency operations. These approaches work very well in taking advantage of certain aspects of the memory hierarchy; however, data movements end up being performed in most cases with the limited view of a *single thread or computational unit*. This scenario can create a myriad of problems when multiple hardware resources are shared between a significant number of threads.

When using compilation techniques to exploit parallelism and locality, tiling is one of the most successful techniques to date. It involves partitioning the iteration space into subdivisions that fit in different levels of the memory hierarchy. Classically, tiling is designed for coarse grain execution to reduce communication overhead. In such designs, parallelism is exploited at the outer tile level and inner tiles run sequentially. Such designs however may lead to idle periods of "slack" that results in resources' under-utilization. To remedy such situation, fine-grain designs to improve locality and parallelism of multithreaded architecture have started to evolve. Designs that take concurrent thread execution into account are able to reduce strain on the entire memory subsystem. However, current designs still overlook the interleaved data movement patterns, irregular access reuse of tiles across caches and memory boundaries that can lead to performance degradation.

As they stand, current software stacks lag behind in exploiting new architectural designs. They need to adapt to these architectural design changes as we head towards massively parallel architectures. Software stacks need to encompass the idea of thread collaboration and orchestration of data movement based on execution patterns of parallel units to fully exploit multithreaded designs. This paper is based on the premise of using hierarchical, tiled loop parallelization techniques to optimize the application access pattern and to better utilize the memory hierarchy of these designs.

In this paper, we present a novel framework whose objective is to improved the use of hardware resources with a primary focus on memory and application's access pattern. Under our framework, tiles are restructured so that the accesses are in contiguous memory space. This ensures that the restructured tiles have minimal interference accessing the same set of data tiles. Thus, with knowledge of the execution pattern and its tile reuse for different levels of the memory hierarchy, we are able to perform timely data movements and operate on tiles in a highly gregarious fashion. Our results show better resource utilization and memory reuse for selected applications as compared to their non restructured counterparts.

978-1-4799-8937-9/15 $31.00 © 2015 IEEE
DOI 10.1109/HPCC-CSS-ICESS.2015.291

The main contribution of this paper is a compiler guided, collaborative data restructuring technique for group reuse based on a dependency and reuse vector. It utilizes the cache hierarchy better by improving access pattern and bringing closely connected elements together. Our technique intelligently shifts the burden between the different caches levels of the cache hierarchy in comparison to the observed default cases. In our case, higher level caches that are closer to the processor suffer less misses whereas further remote caches are subjected to higher strains without impacting negatively memory bandwidth. This burden shift translates into a positive performance outcome.

The rest of the paper is organized as follows. Section II provides motivational examples and the need for the framework presented in this work. Section III presents a description of our fine grained runtime and introduces basic concepts used in our framework. Section IV describes the data restructuring framework, its components and conditions. Section V describes the selected hardware / software testbed and shows the collected data for selected cases. Moreover it discusses the effects and reasons for the performance gains. Section VI provides a snapshot of current efforts in the field to ameliorate the memory access problem. Finally, Section VII provides future work and conclusions for the paper.

II. Motivation

The access pattern of an application significantly affects the performance of its memory subsystem. Data has higher *spatial locality* with contiguous accesses in the address space. Access bandwidth and latencies are sensitive to spatial locality due to memory and cache organization. For example, this is the case when access patterns have smaller strides, yielding accesses that fall into the same memory page or cache line. The end effect are lower page misses, lower bank conflicts and better cache utilization. Similarly, predictable access patterns allow hardware prefetchers to move the required data in advance to their use. Low latency access caches are especially beneficial in cases where the application exhibits plenty of reuse. Strided access pattern by multiple concurrently running threads, however, tax the underlying hardware and quite often lead to access conflicts and misses at various levels of the memory hierarchy. These conflicts lead to premature cache evictions and consequently, high access penalties at every eviction and reuse point.

As an example, consider a classical matrix multiplication as shown in Figure 1. Assuming row major storage, matrix *A* has contiguous access in memory and hence has better spatial locality as compared to matrix *B* access that has a stride of n. Since this example is known to have plenty of access reuse – n^3 times on n^2 data – maximizing reuse in low latency caches to improve performance is of importance. The classical solution to maximize reuse involves tiling for the one-threaded case. In general, though, access penalties arise in loop parallelized code when multiple threads contend over a single tile within the limited confines of the cache. In our example, the parallelized, tiled loop across the *i* dimension shares column tiles in B where the penalty of strided access is paid again and again for each and every access.

```
for (i=1; i<=n;i++)
   for (j=1; j<=n;j++)
      for (k=1; k<=n;k++)
         C[i][j] += A[i][k] * B[k][j];
```

Fig. 1: Matrix Multiplication

In a different scenario, an application can have different access patterns at different points in space and time with multiple references to the same data structure. For example, let us look at an interesting access pattern that can be observed in the LU decomposition code shown in Figure 2. This code factorizes a matrix into lower triangular and upper triangular matrices (and, depending on the variant used, a third "Permutation" matrix) through a series of ever smaller steps. During this process, there are $(n-1)$ rank-one updates of the data matrix, reduced by one in every sweep. The last phase of the update for LU is a matrix multiply operation as seen in the code excerpt in Figure 2. With a blocked version of the LU decomposition code, the execution can be described pictorially by Figure 4. Moreover, every third phase in the figure is a matrix multiply operation on the block. This exposes the same type of issues that plagued the previous matrix multiplication algorithm. This example has the additional property of multiple access patterns. The same data structure is accessed along row and column at the same time in two different locations in *A[i][k]* and *A[j][i]* respectively (visible better in blocked version shown in Figure 3). That means there does not exist one mapping that allows array *A* to be contiguous along all iterations.

A potential solution is to compute a result tile as an outer product and to use an element of A (in the left column sub-block in step 2 of Figure 2) and multiply it with the row of A (in the top row sub-block in step 2) to produce a row of A (in sub-block in step 3). This approach requires writing to the result matrix n times. The more efficient solution would be to use the inner product and calculate the result matrix in one swoop. Figure 4(b) shows a pictorial view of a blocking code where the intermediate block calculation requires the multiplication of row of A (in the left sub block of step 2) and row of A (in the top sub block of step 2). However, the inner product requires strided access for one operand as shown in Figure 3 (inner block calculation in tiled code). Thus, improving locality requires careful analysis of dependencies, access patterns and orchestration of data movement.

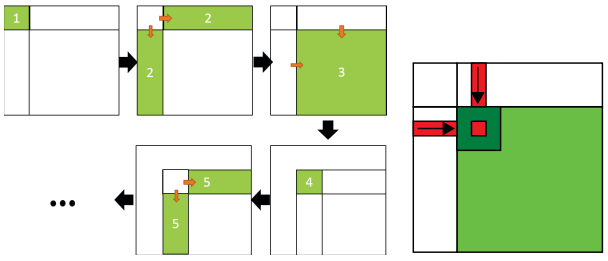
In an effort to ameliorate strided access bandwidth and latency deterioration, we introduce our restructuring framework as described in section IV that takes into account dependencies, access pattern and multiple processing elements working together. Based on the amount of reuse for groups of threads working in close proximity and sharing the same data space, data is reshaped to adapt its access pattern to underlying hardware requirements. For the rest of this paper,

```
for(i=1; i<n;i++){
    for(j=i+1; j<n;j++)
        S1:  A[j][i] = A[j][i]/A[i][i];
        for(j=i+1; j<n;j++)
            for(k=i+1; k<n;k++)
                S2:  A[j][k] = A[j][k]-A[j][i]*A[i][k];
}
```

Fig. 2: LU Decomposition

```
for(jj=16*J;jj<=min(N-1,16*J+15);jj++){
    for(kk=16*K;kk<=min(N-1,16*K+15);kk++){
        range = min(jj-1,min(N-1,16*I+15));
        for(ii=16*I;ii<=range;ii++){
            A[jj][kk]=A[jj][kk]-A[ii][kk]*A[jj][ii];
        }
    }
}
```

Fig. 3: LU Decomposition inner block (16x16) calculation showing row and column access of matrix A to calculate the result matrix as shown in Figure 4(b).



(a) LU Decomposition iterative procedure (b) LU Element in a block calculation

Fig. 4: Iterative Block LU

we will assume that data is stored in row major mapping without a loss of generality. We further limit our presentation to two-dimensional array structures to simplify the discussion, however our technique can also be applied to higher dimensional arrays. We show how our framework restructures data to improve reuse at different levels of the memory hierarchy.

III. Background

In this section, we provide a brief description of a fine-grain execution framework (introduced in [7]) we use in order to reduce communication overhead across threads. Before we delve in the fine grained framework, we need to introduce some terms.

Shrestha et al. [7] proposed a polyhedral based tiling and scheduling technique to exploit locality of parallel threads. In their method, the parallel actors that work on a L2 tile are referred as a "group" and their interactions are limited to the tile and can be influenced to reduce contention, better utilize bandwidth and computational resources. Multiple groups of threads run in parallel exposing both intra and inner tile parallelism. To simplify the execution, they use a fine grain execution framework that represents the hierarchical tiled domain dependencies by a set of bits which are collectively updated by a group of threads working together within the L2 tile. Each thread performs atomic bit-wise operations to create a task mask. Every nonzero bit in the task mask represents a L1 task ready to execute. Such execution happens in a highly parallel fashion and all required updates are done using atomic operations to minimize synchronization overhead. Interested readers are strongly encouraged to read [7], [8] to find the details of this technique.

Our restructuring framework is designed to use such fine-grain execution technique to exploit locality. In addition to the locality gained by grouping threads to reuse data from shared memory space, it reshapes the data to improve execution pattern of all parallel units. In the next section, we explain our restructuring framework. To clarify the concept, we use directional vectors (1,0) and (0,1) to represent direction of mappings, accesses and reuse towards row and column respectively. Also, we use *access matrix* to represent mappings from the iteration space to the data space.

IV. Framework

This section presents our restructuring framework and the different methodologies used to better exploit the memory hierarchy. We present our methodology and the operations needed to restructure the data based on their access patterns.

A. Reuse Analysis

Besides employing reuse matrices *reuse matrix* to perform data movement to software controlled scratchpad memories when reuse is available [9], we also store stride pattern information to optimize memory accesses. Using matrix representation of iteration space and the memory access, we find the amount of reuse.

A rank of a matrix for any statement is given by the number of independent equations, yielding the access dimensionality in our case. Similarly, the depth of a statement is given by the number of nested loops within which the statement resides. When $depth > rank$ [1], it implies that there is $n^{depth} - n^{rank}$ reuse in the iteration space. Revisiting matrix multiplication in Figure 1, n^3 accesses in n^2 data implies that the code has $n^2(n-1)$ reuse.

Continuing the example, the reuse for matrix *A*, *B* and *C* exists along iteration-space vectors (0,1,0), (1,0,0) and (0,0,1) respectively. In case of tiled parallelization at the outermost loop *i*, the reuse vector (1,0,0) is shared by all parallel threads. The reuse exists not only within threads but also across threads. Higher access strides with this kind of shared data (stride n in *B[k][j]*) can potentially slow down all participating threads.

In order to clarify the concept of data sharing between threads, we use the term *self-reuse* whenever there exists a reuse for a set of data that is consumed by the same thread. Similarly, when there exists a reuse for a set of data that is

[1]Both rank and depth have to be in the same domain. For example, a tiled iterator cannot contribute to the depth of a statement.

shared by a group of threads running concurrently, we identify a *group-reuse*. Our goal is to reduce memory latency overhead by reducing the access stride for such data so it fits better in shared caches with minimal conflicts and eviction.

Group-reuse also exists in our LU Decomposition example. In Figure 3, during the final phase of each iterative execution where a tile is computed using matrix multiplication, accesses are strided for one of the input matrix operands. Since this data has reuse throughout the computation of the tiled matrices, latency and bandwidth penalties for strided access have to be paid throughout the execution every time data is evicted and has to be brought into caches.

B. Memory Storage Requirement

We use a straight forward methodology to calculate the memory storage requirement for restructuring. We take the dimension of the iteration-space that is shared and identify strided access reuse across a group of threads. The size of the restructuring space is given by the number of tiles multiplied by the size of a tile in shared dimension.

$$Restructuring\ Space(R) = S * n \tag{1}$$

where S is the size of the tile and n is the number of tiles in the shared dimension.

For example, in Figure 2, the size of the restructuring space for a input matrix of size NxN that is tiled with a tile size $(t * t)$ is given by $t * t * N/t = Nt$.

C. Data Transformation

Based on how the data is accessed, some layouts are better than others. One layout may not provide the best access pattern throughout the execution. In some cases, the same data set can be accessed both by rows and columns within the same iteration space. In other cases, accesses themselves can still be disjoint in time such that at any given time having one mapping is more beneficial. We use information based on the access matrix and its memory mapping and perform data transformations to partial data sets to have access pattern that are more access and cache friendly within a given time frame δt. Such transformation is done in a tiled code such that,

- All elements of outer L2 tiles (i.e the inner L1 tiles in a tiled hierarchy) stay together in memory. The mapping allows memory accesses to take advantage of open memory pages and also results in high residence at various cache levels.
- Elements of the innermost tiles are accessed in a contiguous fashion. This mapping allows better cache locality and reduces unnecessary conflicts.

In order to achieve this, we perform a transformation that allows parallel threads to reuse the allocated restructuring memory space (i.e. increasing the "group reuse" in the application). At different points in time, different disjoint data sets are mapped to the same restructuring space such that there is no conflict. Since our restructuring strategy involves reshaping data to place all elements within a tile together, such transformation requires transformations both at the tile domain and the original domain. One straight forward solution is to find the new indices by traversing the sequence of transformations and compute the modulus for each. This is an expensive proposition. Instead, we compute the displacement (offset used interchangeably) required to map addresses to the restructuring space. This is done in five steps sketched below,

Algorithm 1 currently is tuned to operate under the following two conditions explained below that account for how the tiles are placed in the original space and how the accesses are aligned.

- *Condition 1*: If the shared space (set of outer tiles) is arranged by rows (towards (1,0)) and inner elements are accessed by column (towards (0,1)), then do the transformation with the calculated displacement (Lines 2, 4 and 5 in algorithm 1) and perform a copy transpose (Line 6) at the element level as shown in Figure 5. Initially, *K'* and *J'* are calculated by subtracting lower bounds *K* and 0 respectively. The respective lower bounds become the offset (since they are already relative to the original index *K* and *J*) and are translated to the values corresponding to the original domain by multiplying with the tile size. It is then followed by a transpose at element granularity to change access towards rows.

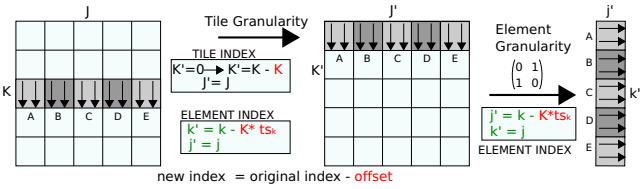


Fig. 5: Transformation Condition 1: Strided access (shown by downward arrows) and row arrangement of parallel tiles (A-E) transformed to have contiguous access using calculated offset (shown in red).

For example, Figure 6 shows how this transformation can be done with our displacement technique. Consider an original 15x15 space and a restructure space of size 3x15 (shown in a red box) where the index (7,4) is mapped to the index (1,4): We calculate the displacement required in all dimensions of the iteration space. In this example, the lower bound in the 'J' and 'K' dimension in the tiled domain is 0 and 2. This can be translated to the original space by multiplying with the number of elements in a tile. Hence the displacement become 0 and 2x3 in the original space in the 'j' and 'k' dimension.

- *Condition 2*: If the outer tiles are arranged by column and the inner elements are accessed by column (towards (0,1)), then do the transformation with the calculated offset (Lines 2,3,4 and 5). This is followed by transpose at element level as shown in Figure 7. Here first *K'* and *J'* are calculated by subtracting the lower bounds and transposing the tile index. The resultant value of *K'* and *J'* i.e 0 and K are then used to calculate the offset relative

Algorithm 1 Data Transformation and Movement

Input: Given array access matrix *A* (accessing towards (0,1)), restructuring space *R*

1: **for** each shared data space partition **do** ▷ set of tiles undergoing data transformation
2: Subtract the lower bound (LBx, LBy...LBn) in a tiled domain for all given dimensions. ▷
3: Transpose in the tiled domain (two innermost dimension) if tile arrangement and accesses are in the same direction (see condition 2 below).
4: Calculate displacement in tiled domain relative to the original tiled index.
5: Translate displacement to reflect the original iteration index (multiplying by tile sizes (tx,ty...tn)).
6: Transpose in original space.
7: **end for**

Output: Indicies mapped to Restructuring Space

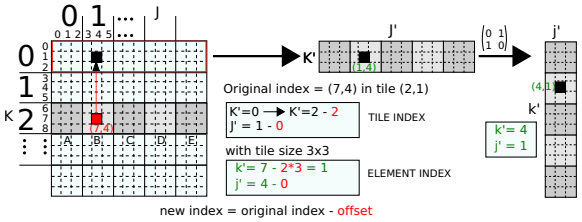


Fig. 6: Transformation example calculating the offset and the new index. First the new index at tile level is calculated relative to the original index K and J, which then is translated to the element index by multiplying them with tile sizes.

to the original index. In this case, relative to the original index *K*, *K'* becomes *K-K* and relative to *J*, *J'* become *J+K-J*. This gives the offset required to calculate the new indices. It is then followed by a transpose at element granularity to change access towards row.

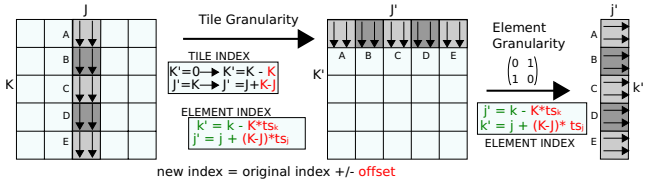


Fig. 7: Transformation Condition 2: Strided access (shown by downward arrows) and column arrangement of parallel tiles (A-E) transformed to have contiguous access using calculated offset (shown in red).

In both cases, the amount of displacement required relative to the original index is calculated to perform targeted transformations. This approach allows calculation of the new indices in a single step without the use of modulus.

D. Exploiting Parallelism

Parallel thread units grab different data set according to their group identification and perform restructuring in a concurrent fashion. Figure 8 and 9 show the code snippet of such displacement done for matrix multiplication (applying condition 2) and LU decomposition (applying condition 1) respectively for tile size 64x64. In both examples, *J* is an iteration towards row and *K* towards column. Parallel units get different iteration of *K* and *J* in matrix multiplication and LU Decomposition respectively to perform restructuring in parallel.

```
offsetk    = K*tilesize;
offsetj    = K*tilesize - J*tilesize;
for (kk=64*K;kk<=min(N-1,64*K+63);kk++)
   for (jj=64*J;jj<=min(N-1,64*J+63);jj++)
      B_RES[jj+offsetj][kk-offsetk]=B[kk][jj];
```

Fig. 8: Matrix Multiplication restructuring data movement

```
offset = K*tilesize;
for (kk=64*K;kk<=min(N-1,64*K+63);kk++)
   for (jj=64*J;jj<=min(N-1,64*J+63);jj++)
      A_RES[jj][kk-offset] = a[kk][jj];
```

Fig. 9: LU Decomposition restructuring data movement

V. Experiments

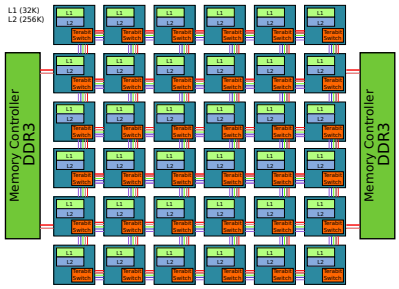


Fig. 10: The Tilera TileGX architecture

Figure 10 shows the pictorial view of our hardware testbed: the Tile-Gx36 architecture. It has 36 processor cores, each equipped with 32KB local 2-way L1 cache and 256KB 8-way L2 cache. All caches are inclusive. Each core also has access to the other core's L2 cache in the grid, giving an impression of a virtual L3 cache. Accesses to L3 caches are much cheaper than accessing the memory (anywhere from 2x to 3x faster). Our

framework uses intratile parallelism to exploit reuse within the grid and performs restructuring for better access strides.

On the software side, we use our running examples, matrix multiplication and LU decomposition, to display the effectiveness of our technique. We also present performance hardware counters to show how our techniques help reduce cache misses. Table I presents the counters we used from TileGX. These counters cover local and remote read and write misses for different cache levels.

Figure 11 shows performance for matrix multiplication over different problem sizes. Our result shows an improvement of up to 26.50% over best case OMP code (OMP_INTRA as in OMP intratile). Additionally, it shows up to 4.5% improvement when compared against our fine-grain grouping techniques briefly discussed in the background section that do not apply restructuring. Figures 12(a), 12(b) and 12(c) show hardware counters for different level cache misses. Our results show reduction in cache misses when compared to the best fine-grain and OMP cases. Compared to the OMP intratile parallel version, fine-grain with restructuring shows reduction in cache misses for all read and write misses at cache levels L1,L2 and L3 ranging from 23.53% to 89.81% except for the remote write misses (aka L3 write misses) where it deteriorates by 138%. Compared to our original work of fine-grain without restructuring, our restructuring policies lead to reduction of local L1 and L2 cache misses by up to 19.1%. However, the remote read and write misses (L3 misses) increase by 12.9% and 15.4% respectively. We believe that increases in remote misses are due to the synchronization overhead required by our framework grouping the physical cores.

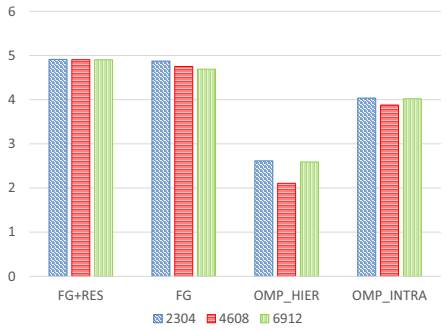


Fig. 11: Matrix Multiplication Performance

Figure 14 shows performance for our second example, LU decomposition, at different problem sizes for fine-grain with/without restructuring and OMP code. For most cases, we use hierarchical and one level tiled OMP code instead of intra-tile OMP parallel code as reference as its performance better. Our technique with restructuring has up to 31.4% advantage over OMP one level tiling and 15.9% advantage over fine-grain without restructuring.

Similarly, Figures 15(a), 15(b) and 15(c) show hardware counters for cache misses. Compare to OMP one level tiling, our technique has an advantage ranging from 17.8% to 72.69%

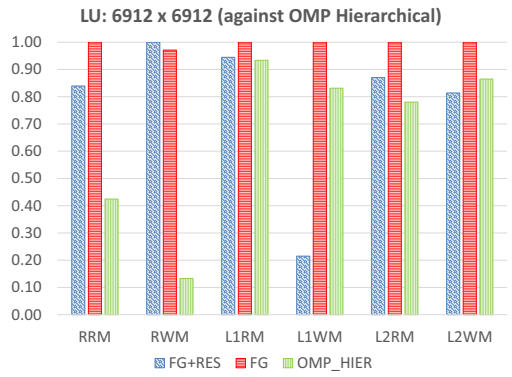


Fig. 13: LU Decomposition Performance counters normalized against OpenMP Hierarchical Implementation

for reduction of all cache misses except for remote L2 writes misses where it increases by 267.23%.

A special case is apparent when looking at the 6912x6912 hierarchical OpenMP case. In this case, our framework performs better than any of the OpenMP cases but it has a perceived disadvantage with almost all the memory hierarchy counters. Our technique has considerable advantage for L1 writes misses and a very small degradation for L1 read misses. These counters are order of magnitude higher than other miss counters and hence contribute to our advantage leading to higher performance (as showcased in Figure 13).

Thus, by doing the restructuring, we are shifting work from the higher levels of the memory hierarchy toward the lower levels and increasing the locality of the levels closest to the processor, thus, improving overall performance. However, we have a trade-off between the framework overhead that comes from the interplay between the framework, the actual buffer and certain caches features (e.g. the inclusiveness) as showcased in the performance counters. Even then, the performance improvements are still substantial.

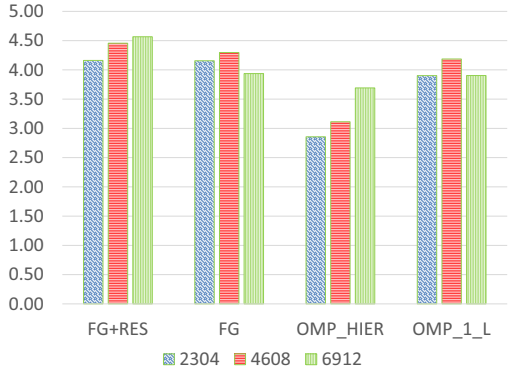


Fig. 14: LU Decomposition Performance

In summary, by judicious data allocation and data restructuring guided by compiler driven data reuse analysis, we are able to shift the burden between the different cache level

Performance Counter	Description	Short-hand
READ_MISS	Level 1 Cache misses for reads	L1RM
WRITE_MISS	Level 1 Cache misses for writes	L1WM
LOCAL_DATA_READ_MISS	Local Level 2/3 Cache misses for reads	L2RM
LOCAL_WRITE_MISS	Local Level 2/3 Cache misses for writes	L2WM
REMOTE_DATA_READ_MISS	Remote Level 2/3 Cache misses for reads	RRM
REMOTE_WRITE_MISS	Remote Level 2/3 Cache misses for writes	RWM

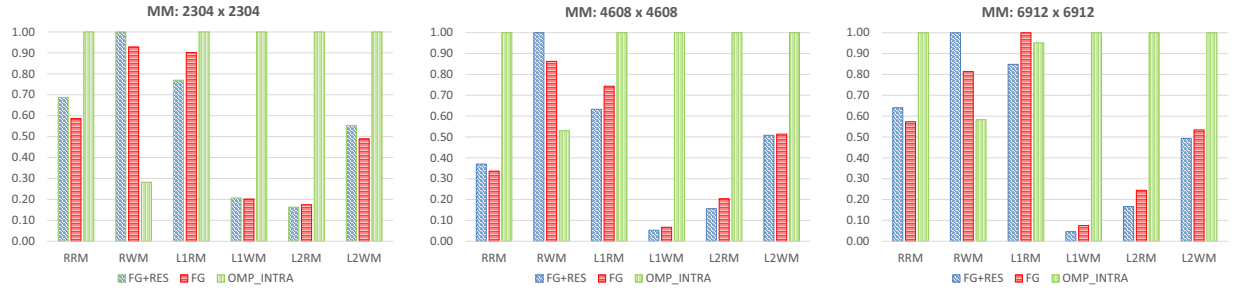
TABLE I: Performance Counters collected in TileGx36

Size	GFlops	RRM	RWM	L1RM	L1WM	L2RM	L2WM
FineGrain+Restructuring							
2304	**4.9119**	9.20E+07	1.18E+08	1.67E+10	1.34E+08	3.62E+06	5.29E+05
4608	**4.9051**	7.17E+08	2.84E+08	1.39E+11	3.37E+08	2.91E+07	3.87E+06
6912	**4.9029**	2.37E+09	6.33E+08	4.68E+11	8.12E+08	1.01E+08	1.25E+07
FineGrain							
2304	4.873	7.86E+07	1.09E+08	1.95E+10	1.30E+08	3.88E+06	4.68E+05
4608	4.7514	6.51E+08	2.45E+08	1.63E+11	4.26E+08	3.80E+07	3.91E+06
6912	4.6898	2.12E+09	5.15E+08	5.52E+11	1.33E+09	1.48E+08	1.35E+07
OMP_Hierarchical							
2304	2.6142	1.42E+08	1.27E+10	1.85E+10	1.38E+10	4.11E+06	4.05E+05
4608	2.1063	2.67E+09	1.01E+11	2.32E+11	1.10E+11	7.42E+07	3.40E+06
6912	2.5894	4.76E+09	3.41E+11	5.06E+11	3.73E+11	1.71E+08	1.42E+07
OMP_with_Intratile_Parallelism							
2304	**4.0357**	1.34E+08	3.33E+07	2.17E+10	6.46E+08	2.22E+07	9.56E+05
4608	**3.8775**	1.94E+09	1.50E+08	2.19E+11	6.37E+09	1.86E+08	7.62E+06
6912	**4.0186**	3.70E+09	3.69E+08	5.25E+11	1.77E+10	6.08E+08	2.53E+07

TABLE II: Performance Counters collected in TileGx36 for Matrix Multiplication. Bold green and blue values represent the best values for Fine grain and OpenMP experiments

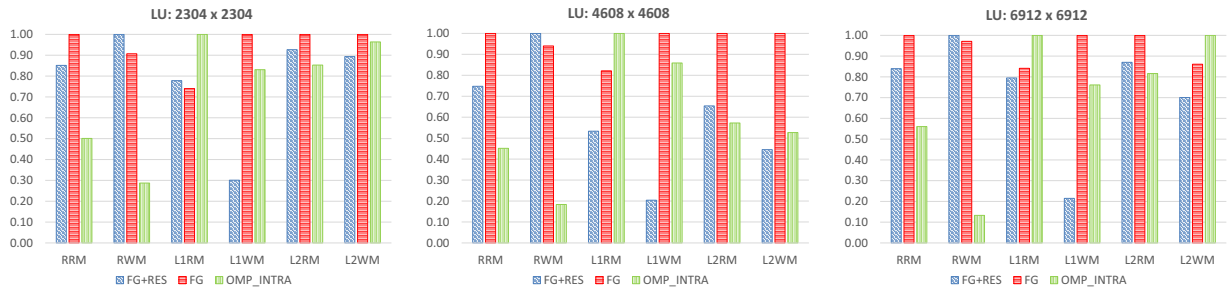
Size	GFlops	RRM	RWM	L1RM	L1WM	L2RM	L2WM
FineGrain+Restructuring							
2304	**4.1616**	1.13E+08	5.83E+07	6.02E+09	8.29E+07	1.93E+06	1.99E+05
4608	**4.4568**	9.52E+08	3.55E+08	4.88E+10	5.01E+08	1.56E+07	1.18E+06
6912	**4.5659**	2.70E+09	1.06E+09	1.67E+11	1.65E+09	5.07E+07	3.40E+06
FineGrain							
2304	4.1562	1.33E+08	5.29E+07	5.72E+09	2.76E+08	2.08E+06	2.22E+05
4608	4.297	1.27E+09	3.33E+08	7.51E+10	2.46E+09	2.39E+07	2.65E+06
6912	3.9364	3.22E+09	1.03E+09	1.77E+11	7.70E+09	5.82E+07	4.18E+06
OMP_Hierarchical							
2304	2.8563	5.06E+07	1.52E+07	6.20E+09	2.37E+08	1.69E+06	1.64E+05
4608	3.1133	5.77E+08	6.59E+07	8.09E+10	2.11E+09	1.77E+07	8.95E+05
6912	3.6915	1.36E+09	1.41E+08	1.65E+11	6.40E+09	4.54E+07	3.62E+06
OMP_One_Level							
2304	**3.9019**	6.67E+07	1.67E+07	7.73E+09	2.29E+08	1.77E+06	2.14E+05
4608	**4.1863**	5.75E+08	6.51E+07	9.14E+10	2.11E+09	1.37E+07	1.40E+06
6912	**3.9029**	1.80E+09	1.41E+08	2.10E+11	5.86E+09	4.75E+07	4.86E+06

TABLE III: Performance Counters collected in TileGx36 for LU Decomposition. Bold green and blue values represent the best values for Fine grain and OpenMP experiments



(a) MM 2304 (b) MM 4608 (c) MM 6912

Fig. 12: Matrix Multiplication performance and hardware counters normalized to the maximum value for that counter



(a) LU 2304 (b) LU 4608 (c) LU 6912

Fig. 15: LU performance and hardware counters normalized to the maximum value for that counter

hierarchies, yielding better performance outcomes for our two very relevant test cases.

VI. Related Work

Due to its importance, data movement and locality is the target of substantial compiler and runtime research. Data centric optimizations in multithreaded environments have become more visible. There is a growing trend towards giving programmers control of data management in order to improve performance. The common approach uses information about access pattern, strides, data sharing and reuse to reduce memory acess latency.

Overlapping data movement with computation and data prefetching based on access pattern are typical data movement orchestration strategies. Garcia [10] used percolation along with dynamic scheduling to improve the locality of dense matrix multiplications on the Cyclops-64 manycore architecture. He used helper threads to perform explicit data movement to a double buffered SRAM. Baskaran [9] used explicit data movement and index transformations to improve locality and reuse on scratchpads. He developed an automatic polyhedral based framework to utilize the available processing resources and manage data movement to hide memory access latency. This technique seems very efficient on exploiting parallelism and managing data movement on GPUs where accesses happen in SIMD fashion.

Data transformation work has also been done to improve vectorization by Jang [11]. Based on the access pattern, different data transformation rules are used to have contiguous access, enabling vectorization. Jeremiassen [12] showed that false sharing misses can be reduced by performing data transformations. It uses a transpose when adjacent elements are accessed by different processors, uses indirection when data restructuring is not possible and padding shared data writes when falsely shared. Also, Rivera [13] showed how cache conflicts can be avoided using padding techniques. Lu [14] used data layout transformations using the polyhedral model to reduce hot spots and bank contention.

Prior work has also showed that array restructuring can be used to improve spatial locality in cases when loop transformation is not enough. Leung [15] with his restructuring technique allowed transformation of data to provide the better access pattern. technique is independent of loop carried dependencies, providing flexibility in the application of this transformation. However, careful consideration is needed to make sure this transformation is not detrimental when the same memory structure is accessed in multiple ways. Non-canonical data layouts [16], [17], [18], [19] have also been used to reduce memory access latency. Indeed, our approach also uses non-canonical layouts, however we do so to improve residence for the caches closest to the processor – sometimes at the expense of the farther remote caches – when reuse exists across parallel threads.

In some cases, both, loop and data transformations, are not able to produce efficient code independently. Cierniak and Li [20] proposed a technique to unify these two techniques using mapping and stride vectors. Within the loop nest, different array structures can use different mapping based on access pattern, allowing contiguous access in memory space. Although their techniques are difficult in cases of multiple references, single reference access transformations work well. Also, Kandemir [21] proposed an integrated framework that uses loop transformation for temporal locality and data transformation for spatial locality.

One of the issues applying data transformations is the extra memory space requirement. Darte [22] proposed a mathematical framework that maps indexes to a limited memory space. The strategy allows effective use of shared space, like e.g. scratchpad, and ensures conflict free mapping by avoiding conflicting elements with simultaneous live. Also, such a technique can be used to hold intermediate values and maximize reuse. We believe that our framework can take advantage of such technique for optimizing restructuring space.

Data centric optimizations work have also been done without involving data restructuring. Kodukula [23] used data blocking based on its flow through the memory hierarchy to optimize memory access. His approach selects a sequence of blocks that is touched by a processor and executes statements associated with those blocks. Nevertheless, instances with complex programs structures and dependencies make this transformation approach unwieldy.

Most of the work presented here in this related section focuses mainly on a single thread of execution and overlooks the opportunity of data movement and restructuring that can benefit a group of thread working gregariously in close proximity in time and space. Our work attempts to optimize accesses that enable threads to work together ,improving the access pattern of shared data where all participating threads contribute to improve the utilization of a memory subsystem.

VII. Future work and Conclusions

We have developed a thread collaborative restructuring strategy, within a hierarchical loop transformation framework, that makes memory accesses more efficient. As part of our future work, we plan to investigate the interplay between the cache's properties and the restructuring space. For our current experiments, the inclusiveness of the TileGX cache might result in the disproportional increase of remote write misses (L3 write misses). Moreover, the inclusion of hardware support to help with the restructuring effort (such as DMA engines, gather and scatter form and to memory, etc) might increase the performance of this effort considerable. In addition, we need to characterize the effects of prefetching hardware and its interplay with the restructuring buffer. A larger set of architectural features, application codes and data set sizes is needed to better characterize the effects of the restructuring buffers.

With current many cores providing processing resources in abundance, shared resource like memory and bandwidth are still relatively scant. It stands to be true that memory access latency has been one of the major performance bottlenecks. Our technique provides a collaborative view of data management, restructuring and execution to improve locality for all participating parallel threads. We believe such techniques are essential to improve the utilization of the memory subsystem and reach next milestone in performance.

VIII. Acknowledgments

This research was supported in part by DOE ASCR XStack program under Awards DE-SC0008716, DE-SC0008717 and NSF XPS 1439097.

References

[1] M. Wolfe, "Iteration space tiling for memory hierarchies," in *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, (Philadelphia, PA, USA), pp. 357–361, Society for Industrial and Applied Mathematics, 1989.

[2] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, Supercomputing '89, (New York, NY, USA), pp. 655–664, ACM, 1989.

[3] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, (New York, NY, USA), pp. 30–44, ACM, 1991.

[4] D. Kim, S. S.-w. Liao, P. H. Wang, J. d. Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen, "Physical experimentation with prefetching helper threads on intel's hyper-threaded processors," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, (Washington, DC, USA), pp. 27–, IEEE Computer Society, 2004.

[5] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors," *Siam Review*, December 2008.

[6] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, (Washington, DC, USA), pp. 285–, IEEE Computer Society, 1999.

[7] S. Shrestha, J. Manzano, A. Marquez, J. Feo, and G. Gao, "Jagged tiling for intra-tile parallelism and fine-grain multithreading," in *Languages and Compilers for Parallel Computing* (J. Brodman and P. Tu, eds.), vol. 8967 of *Lecture Notes in Computer Science*, pp. 161–175, Springer International Publishing, 2015.

[8] S. Shrestha, G. R. Gao, J. Manzano, A. Marquez, and J. Feo, "Locality aware concurrent start for stencil applications," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, (Washington, DC, USA), pp. 157–166, IEEE Computer Society, 2015.

[9] M. M. Baskaran *et al.*, "Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 1–10, ACM, 2008.

[10] E. Garcia, D. Orozco, R. Khan, I. E. Venetis, K. Livingston, and G. R. Gao, "Dynamic percolation: A case of study on the shortcomings of traditional optimization in many-core architectures," in *Proceedings of the 9th conference on Computing Frontiers*, pp. 245–248, ACM, 2012.

[11] B. Jang, P. Mistry, D. Schaa, R. Dominguez, and D. Kaeli, "Data transformations enabling loop vectorization on multithreaded data parallel architectures," in *ACM Sigplan Notices*, vol. 45, pp. 353–354, ACM, 2010.

[12] T. E. Jeremiassen and S. J. Eggers, *Reducing false sharing on shared memory multiprocessors through compile time data transformations*, vol. 30. ACM, 1995.

[13] G. Rivera and C.-W. Tseng, "Data transformations for eliminating conflict misses," in *ACM SIGPLAN Notices*, vol. 33, pp. 38–49, ACM, 1998.

[14] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, *et al.*, "Data layout transformation for enhancing data locality on nuca chip multiprocessors," in *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pp. 348–357, IEEE, 2009.

[15] S.-T. Leung and J. Zahorjan, *Optimizing data locality by array restructuring*. Department of Computer Science and Engineering, University of Washington, 1995.

[16] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam, "Data and computation transformations for multiprocessors," in *ACM SIGPLAN Notices*, vol. 30, pp. 166–178, ACM, 1995.

[17] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi, "Recursive array layouts and fast matrix multiplication," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 11, pp. 1105–1123, 2002.

[18] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi, "Nonlinear array layouts for hierarchical memory systems," in *Proceedings of the 13th international conference on Supercomputing*, pp. 444–453, ACM, 1999.

[19] S. Chatterjee and S. Sen, "Cache-efficient matrix transposition," in *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, pp. 195–205, IEEE, 2000.

[20] M. Cierniak and W. Li, "Unifying data and control transformations for distributed shared-memory machines," in *ACM SIGPLAN Notices*, vol. 30, pp. 205–217, ACM, 1995.

[21] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, "Improving locality using loop and data transformations in an integrated framework," in *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pp. 285–297, IEEE Computer Society Press, 1998.

[22] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *Computers, IEEE Transactions on*, vol. 54, no. 10, pp. 1242–1257, 2005.

[23] I. Kodukula, N. Ahmed, and d Keshav Pingali, "Data-centric multi-level blocking," pp. 346–357, 1997.