2011 International Conference on Parallel Architectures and Compilation Techniques

Enhancing Data Locality for Dynamic Simulations through Asynchronous Data Transformations and Adaptive Control

Bo Wu, Eddy Z. Zhang, Xipeng Shen
Department of Computer Science
The College of William and Mary, Williamsburg, VA 23187
{bwu,eddy,xshen}@cs.wm.edu

***Abstract*—Many dynamic simulation programs contain complex, irregular memory reference patterns, and require runtime optimizations to enhance data locality. Current approaches periodically stop the execution of an application to reorder the computation or data based on the current program state to improve the data locality for the next period of execution. In this work, we examine the implications that modern heterogeneous Chip Multiprocessors (CMP) architecture imposes on the optimization paradigm. We develop three techniques to enhance the optimizations. The first is *asynchronous data transformation*, which moves data reordering off the critical path through dependence circumvention. The second is a novel data transformation algorithm, named *TLayout*, designed specially to take advantage of modern throughput-oriented processors. Together they provide two complementary ways to attack a benefit-overhead dilemma inherited in traditional techniques. Working with a *dynamic adaptation scheme*, the techniques produce significant performance improvement for a set of dynamic simulation benchmarks.**

I. Introduction

Due to the memory wall problem on traditional architecture, data locality has been one of the most prominent factors that determine the performance of a program. Its importance is even more pronounced on modern Chip Multiprocessors (CMP), where, the last-level cache and memory bus bandwidth are typically shared by multiple cores. The sharing causes contention among co-running applications, and the effect intensifies as the number of cores grows on a chip. Data locality enhancement is an important approach to tackling the problem. It reduces the required accesses to the last-level cache and memory to alleviate the pressure on shared memory hierarchy.

Special difficulties for locality enhancement come from irregular memory references. Such references often arise in dynamic simulation applications—such as unstructured mesh simulation and molecular dynamics simulation—due to the use of sparse and irregular data structures. A representative form of irregular references is $A[P[i]]$, where the index array P may be embodied in real applications by an input array or intermediate computation results that are difficult to know until run time.

Irregular memory references have several properties. First, because the content of the index array P may be arbitrary, the references to A tend to cause serious locality issues. Second, the memory access patterns of those references are unknown until execution time. Third, applications having such references tend to contain a main loop—such as, the mesh refinement loop in mesh generation, the time elapse loop in molecular dynamics simulation—that encloses the irregular memory references. The access patterns of the references (e.g., the values in P) often vary across the loop iterations. These properties make locality enhancement of irregular references extremely difficult for static compilation techniques.

A number of prior studies [8], [12], [21], [22], [26] have pursued runtime data transformations to attack dynamic irregular references. The strategy is to reorder data objects *during an execution* based on their exhibited access patterns.

However, the power of the prior transformations has been restrained by a dilemma. In all prior techniques, the runtime data or computation reordering happens synchronously—that is, the reordering is on the critical path of the application. This feature results in a tension between transformation quality and runtime overhead: More sophisticated transformations often yield better locality and save more execution time, but at the same time, they add more transformation overhead to the overall execution. The overhead can be substantial, especially for sophisticated transformations. For instance, one application of RCB—a classic data transformation approach—takes more than 20 simulation time steps in most experiments reported in Section VII. Moreover, the transformation have to be applied repetitively due to the iterative computations in dynamic simulations. Some studies propose to apply the transformation occasionally rather than everytime when access pattern changes [11], [20]. Unfortunately, it is subject to the same quality-overhead dilemma: The less frequently the transformation applies, the less overhead it causes, but the worse the data layout is.

In this paper, we propose three orthogonal techniques to resolve the quality-overhead dilemma.

The first is *asynchronous data transformation*, supported by a dependence-circumventing decomposition. The basic idea is to hide the transformation overhead by offloading the main transformations from the critical path, making them happen asynchronously (on an idle processor) in parallel with the execution of the application. Despite the simplicity of the idea, to the best of our knowledge, asynchronous data transformation has not been proposed previously. The plausible reason exists in the circular data dependences between data transformations and the execution of the application. On one hand, the transformation modifies the data structure that the application needs to read; on the other hand, the transformation needs to read some results computed by the application to figure out the appropriate data order. So, inherently, one invocation of a data transformation must run serially with the corresponding iteration of the application. In this work, we circumvent the problem by decomposing data transformation into two parts and safely relaxing some dependences through a careful analysis and layout approximation.

The second technique we develop aims at overhead minimization, especially for a system equipped with massive parallel devices (e.g., GPU). We propose a novel data transformation algorithm, named *TLayout* (*T* for throughput), which reduces transformation overhead significantly with little compromise to the resulting quality. Unlike traditional data transformation algorithms, TLayout is a massively data-parallel algorithm, specially customized to the strengths of throughput-oriented co-processors. It is novel in using an almost dependence-free approach to grouping nodes into a number of clusters such that the nodes referenced adjacently fall into the same cluster. The algorithm shows high efficiency and scalability.

Asynchronous data transformation and TLayout tackle the limitations of previous data transformations in two orthogonal directions; one for overhead hiding, the other for overhead minimization. Together, they help resolve the quality-overhead dilemma that prior approaches have been facing.

The third technique we develop is an online adaptive scheme. By transparently selecting the appropriate transformation strategy during runtime, the scheme gains the best of both asynchronous and synchronous transformations, proving able to overcome the limitations of both strategies.

Overall, the proposed techniques yield 65% higher performance improvement than previous techniques do, accelerating the original dynamic simulations by as much as a factor of 3.1 (2.4X on average) on five representative dynamic simulation benchmarks.

In summary, this work makes four main contributions:

- To the best of our knowledge, this work is the first study that proposes and implements asynchronous data transformation for enhancing data locality of irregular dynamic simulations. The technique resolves the quality-overhead dilemma that has been limiting the effectiveness of prior runtime data transformations.
- It presents a new data transformation algorithm, TLayout, which, as far as we know, is the first algorithm designed to take advantage of the massive parallelism of through-oriented processors for enhancing data locality of CPU applications.
- It introduces an online adaptive scheme to safely exert the power of both synchronous and asynchronous transformations.
- This study initiates a new type of collaborations between CPU and co-processors. We are not aware of any previous proposals of using co-processors to do runtime program optimizations for CPU applications. This new direction may lead to some unconventional ways to exploit the power of heterogeneous computing systems.

II. Background on Irregular References and Runtime Locality Enhancement

Irregular references commonly exist in dynamic simulation programs due to the use of sparse and irregular data structures. They are typically in forms of indirect references like $A[P[i]]$.

Previous solutions to irregular references use runtime data and computation reordering. In computation reordering, the iterations of the central computation loop are reordered so that the iterations accessing the same or adjacent data elements are adjacent in time. This transformation requires that there are no dependences across the iterations. Techniques for determining the suitable iteration order include lexicographical sort [6], bucket sort [22], z-sort [12], and so on.

Data reordering repositions elements in an array to improve spatial locality. The basic strategy is to relocate the elements such that the elements that tend to be accessed closely in time become close in memory space. Because determining optimal data orders is an NP-hard problem in general [23], researchers have proposed various heuristics-based algorithms, including consecutive packing (CPACK) [8], Reverse Cuthill-McKee (RCM) [18], space filling curve (SFC) [21], recursive coordinate bisection (RCB) [4], multi-level graph partitioning (METIS) [14], and hierarchical clustering algorithm (GPART) [12]. Previous studies [8], [12] have found that in most cases, the combination of the two—a data reordering followed by a computation reordering—gives better results than each alone. In the following discussion, we use *data transformation* to refer to the transformations that use data reordering or/and computation reordering for locality enhancement.

```
for each time step
  if time_to_update()
    IList = update_IList (Location);
  end if

  /* main computation with irreg. references to Location */
  for each (i,j) in IList
    f = calculate_force (Location[i], Location[j]);
    Force[i] += f;
    Force[j] -= f;
  end for

  for each particle i
    Location[i] = update_loc (Location[i], Force[i]);
  end for
end for
```

Fig. 1. The main loop of Moldyn.

In all prior research, data transformation is applied synchronously with the application. It is placed on the critical path of the application execution, hence subject to the quality-overhead dilemma mentioned in Section I.

III. Asynchronous Data Transformation

Asynchronous data transformation is our first technique for resolving the dilemma between transformation quality and overhead. The basic idea is simple: putting data transformation on a helper processor so that it can happen in parallel with the application execution. However, to the best of our knowledge, this simple idea has never been realized before. A plausible reason for the absence is the inherent data dependences between data transformation and the transformed application. To help explanation, we first outline the sketch of a dynamic simulation program, Moldyn, as our example.

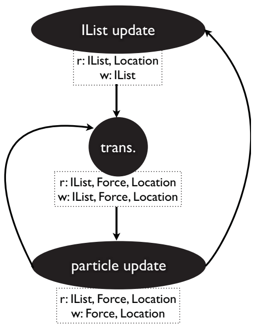
A. An Example Irregular Dynamic Simulation Program

Moldyn is a program for simulating the movements of many particles caused by their interactions. The program maintains a list, named "interaction list", to record the particles that are close enough to interact with each other. The list consists of a number of pairs; each pair contains the IDs of two particles that are close enough in the particle space to have interactions. Figure 1 shows the pseudo-code of the computation kernel of Moldyn. It contains a time-step loop. In each iteration, the program first checks if it is time to update the interaction list *IList*; if so, it makes the update based on the current locations of the particles. It then traverses the interaction list, and computes the force that a particle receives from its neighbors. After that, the program updates the locations of each particle based on the newly computed forces.

Apparently, the major computation is on the force calculation loop. The references to the *Location* and *Force* arrays in that loop are irregular references; the *IList* array plays the role of an index array, whose content decides which elements of *Location* and *Force* are referenced at which iteration of the loop. Each time when *IList* gets updated, the patterns of the references to *Location* and *Force* change accordingly.

This example shows some representative features of irregular dynamic simulations. These applications usually contain a main loop (e.g., the time-step loop) that encloses irregular references. The irregular references involve two data structures; one is the *reference target* (e.g., *Location* and *Force*), the other is the *reference clue* (e.g., *IList*)[1]. The values of the reference clue often vary across the main loop iterations.

[1]These terms are similar to "index array" and "data array" in some earlier work; using them helps avoid confusion with some other terms in this paper.



```
for each time step
  if time_to_update()
    IList = update_IList (Location);
    dataTrans(IList,Location,Force);
  end if
  ... ...
end for
```

Fig. 2. Dependence graph (left) and the synchronous data transformation (right) for the Moldyn example. ("r" and "w" lists the sets of data that are read and written respectively.)

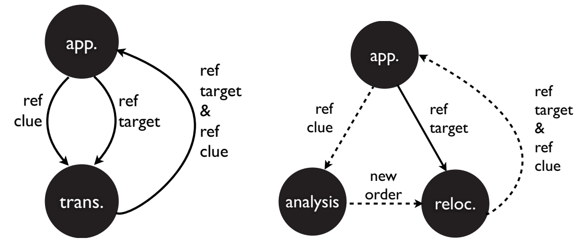


Fig. 3. Dependences between data transformation and the application. Each edge is a data dependence edge labeled with the related data. Broken edges show dependences that are relaxed in asynchronous data transformation.

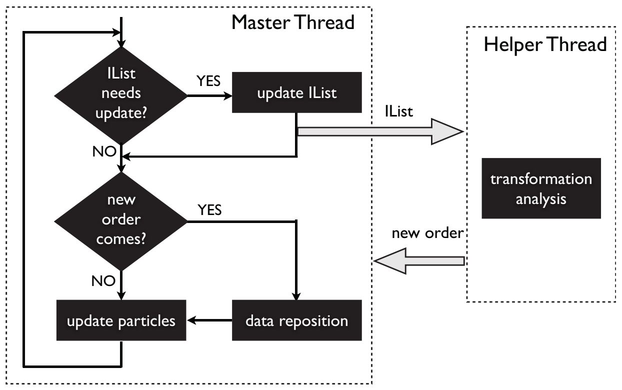


Fig. 4. Control flow of asynchronous data transformation for Moldyn.

B. Synchronous Data Transformations

Runtime data transformation can benefit the force calculation loop in the Moldyn example. The basic strategy of a typical transformation is to reorder the items in the reference target according to the content of reference clue—for example, moving particles that have interactions (by reading *IList*) close to one another in *Location* and *Force* in the Moldyn example. Very often, a following computation reordering is applied as part of the data transformation, in which, the iterations of the loop (e.g., the force computation loop) that encloses the irregular references are reordered. For the Moldyn example, it can be realized by reordering the pairs in *IList* based on the new order of particles to further improve the locality.

One place to put the transformation is between the update of the reference clue and the accesses to the reference target, as shown in Figure 2. This placement is natural because of the data dependences among those components. In fact, this placement is what prior studies adopt. Because the data transformation is put on the critical path of the execution, we call it *synchronous data transformation*.

C. Decomposition and Dependence Relaxation

Data dependences between data transformation and the application form a major obstacle for asynchronous data transformation. Figure 3 (a) summarizes the bi-directional (true) dependences.

We circumvent the dependences based on two properties of data transformations. The first is that most data transformations can be decomposed into an order analysis step and a data relocation step. The order analysis step computes a locality-favorable order according to the reference clue, and the relocation step repositions items in the reference target (and reference clue) based on the produced order. When a data transformation is decomposed into these two components, the two dependence edges from the application to the transformation become pointing to the two components respectively, as Figure 3 (b) shows.

The second is that not all dependences between data transformations and the application are critical. Among the four dependences shown in Figure 3 (b), the dependence from the application to the analysis component is not critical for the correctness of the execution. In another word, if we violate the dependence, the produced locality-favorable order may not lead to a desirable layout for the reference target, but the execution of the application will be correct still. Similarly, if we violate the dependence from the relocation component to the application, the application may have to use an old layout of the reference target rather than the enhanced one; it may hence run slower than it could, but will still produce the correct result. The same is true for the dependence from the analysis component to the relocation component. On the other hand, the dependence from the application to the relocation component is critical. A violation of this dependence may cause the transformed reference target (e.g., the *Location* and *Force* arrays) to contain obsolete values and impair the correctness of the execution.

Based on the two properties, we develop asynchronous data transformation by relaxing the three non-critical dependences in Figure 3 (b). The key of the implementation is to decompose data transformation into two components, leave the relocation component on the critical path but make the analysis component run by a helper thread asynchronously, and allow the use of obsolete reference clues for the computation of new data orders.

Figure 4 outlines the basic control flow for the Moldyn example. The master thread executes the application and the relocation component, while the helper thread runs the analysis component in parallel. At an update to the interaction list, the master thread sends the new *IList* (or some other reference clue, e.g., coordinates of nodes) to the helper thread, and then continues its execution while the helper thread computes for a new locality-favorable data order. If the new order is not ready yet when the master thread reaches the "new order ready?" check, it continues executing the following part of the application using current data layouts. When the helper thread finishes computing the order (several time steps may have passed since the order computation starts), it sets a flag so that when the master thread reaches the "new order ready?" check again, it can use the new order to reposition the reference target and reference clue to improve the locality of some following iterations.

This design makes the analysis component of data transformation proceed asynchronously with the application, but leaves the reposi-

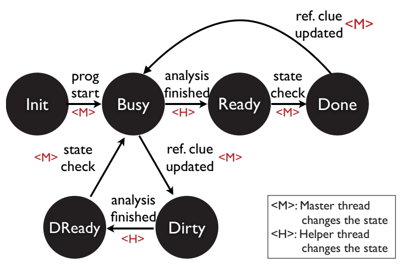


Fig. 5. State transitions for thread coordination in asynchronous data transformation.

tion component on the critical path. In many data transformations, the most costly part is in the order analysis rather than the data repositioning–as Figure 8 will show, the time ratios between them are between 6:4 and 8:2 for RCB. This design hence hides the majority of the data transformation overhead. Meanwhile, because the placement of reposition component maintains the critical dependence (the solid line in Figure 3 (b)), the application still runs correctly.

We now examine how the asynchronous data transformation relaxes the three non-critical data dependences (the broken lines in Figure 3 (b)), and the consequences. For the dependence from an application to the analysis component, in the asynchronous transformation, the reference clue passed to the analysis component may be obsolete. It happens when the order analysis takes longer time than an update period of the reference clue. As a result, the new order passed from the analysis component to the repositioning component may be not as good as the computed data order if the current reference clue was used. The ultimate consequence is that the layouts of the reordered reference target and reference clue fit the obsolete rather than the current reference clue well. This analysis reveals a potential loss of the data transformation benefits incurred by the asynchronous scheme. Section VI will show how this loss can be largely prevented.

D. Thread Coordination

In this part, we present some implementation details on supporting the coordination between a master thread and a helper thread in asynchronous data transformation. The implementation is based on a 6-state transition graph to ensure in-time data transfers and meanwhile avoid unnecessary data copies.

We use a shared variable, protected by a lock, to coordinate the master thread and the helper thread. Figure 5 shows the states recorded by the variable and the state transitions.

When an execution starts and a helper thread is created, the master thread sends the current reference clue to the helper thread, and sets the state to "busy". From the "busy" state, there are two circular paths.

- *Bottom circular path.* When the master thread finishes an update to the reference clue and the state is still "busy", it changes the state to "dirty", indicating that a new order needs to be computed because the reference clue has changed. When the helper thread finishes its current job and passes its computed order to the master thread, it changes the state from "dirty" to "dready". At the next state check by the master thread, it will see that the helper thread has just prepared a new data order and also it needs to get the current reference clue to compute another data order. The master thread then sends the current reference clue to the helper thread, conducts a reposition transformation using the new order, and then changes the state to "busy".
- *Right circular path.* If the helper thread finishes its job within one update period of the reference clue, it changes the state to "ready". At the next state check by the master thread, it will see that the helper thread has just prepared a new data order. It conducts the reposition transformation and then changes the state to "done". Note that it does not send the current reference clue to the helper thread because the clue is identical to what the helper thread already has, which is the key difference between the "read" and "dready" states. At the next update of the reference clue, the master thread sends the clue to the helper thread and changes the state to "busy".

The design of the state transitions ensures that both threads receive necessary data in time, and meanwhile avoids unnecessary data transportation. For instance, consider a case where during the computation of one new data order, the master thread updates the reference clue three times. The state will remain "dirty" after the first update until the helper thread finishes its job. As the master thread sends no data in the "dirty" state, only the most recent reference clue (i.e. the one after the third updates) is sent to the helper thread.

Asynchronous data transformation hides most transformation overhead, but is subject to the use of obsolete reference clue. The longer a transformation takes, the more obsolete the used reference clue is. Even though in many cases, the gain exceeds the loss as Section VII will show, reduction of the transformation time will make its benefit more pronounced—the goal of the technique presented next.

IV. TLAYOUT: A TRANSFORMATION ALGORITHM FOR THROUGHPUT-ORIENTED PROCESSORS

The second technique we develop is *TLayout*, a data transformation algorithm for reducing transformation overhead by exploiting the special features of throughput-oriented processors.

The motivation comes from the trend in modern architecture development. Due to the high throughput and power efficiency, throughput-oriented processors (e.g., GPU) are being increasingly adopted to co-run with general-purpose CPUs. This trend is underscored by the recent Intel Sandy Bridge and upcoming AMD Fusion processors, which have CPU and GPU on a single chip. Exploiting throughput-oriented co-processors for irregular applications is a challenge, given that these massively parallel co-processors are typically weak in handling computations with complex memory references, dependences, and control flows.

Our idea is to use such co-processors to accelerate data transformations for CPU executions. This use of the co-processors is especially appealing for legacy CPU code, because it needs virtually no code changes. Programmers only need to insert three function calls (see Section V) to invoke a data transformation function we have developed for the co-processors. In contrast, many efforts are needed for porting and tuning an irregular application to co-processors [28], [29]. The paradigm of using co-processors for program optimizations offers an *easy, quick* way for legacy programs to benefit from the co-processors (even though the performance from manual code porting may be higher).

Unfortunately, none of previous data transformation algorithms is designed for massively parallel architectures. Their complex control flows and dependences make them unsuitable for throughput-oriented processors.

TLayout Algorithm: TLayout is our solution to the problem. It is designed to be massively data parallel. It produces locality of the similar quality as sophisticated classic transformation algorithms do, but with one third of the overhead (on GPU).

As with many previous data transformation algorithms [12], TLayout is based on the underlying graph structure of data references in the application. Simply speaking, data elements that are referenced closely (e.g., in one iteration of an inner loop) are regarded as neighbor nodes in a reference graph, having an edge in between. Data locality optimizations are then mapped to a graph partitioning problem. Partitioning the graph and putting nodes in a partition close in memory usually improves spatial and temporal locality. In dynamic simulation programs, the reference graphs are often already embedded in the reference clue—such as the interaction list in Moldyn, and the mesh structure in a mesh refinement application. As the reference graphs of these applications usually come from the spatial or topological relations among objects (e.g., particles in a physical space), it is typical that one reference graph covers all interesting data objects.

The strategy of TLayout is incremental clustering through iterative membership-propagation based on the topology of the reference graph. The input to TLayout is a reference graph, encoded as a number of node pairs, with each pair consisting of two nodes that are connected by an edge in the reference graph. The output is a number of clusters that partition the nodes of the graph completely in a way that the nodes close in topology belong to the same cluster. The algorithm starts by setting the membership of each node (i.e., which cluster it belongs to) to *null*. It then proceeds in the following steps:

1) SEED PLANTING: TLayout randomly selects K nodes as the seeds for K clusters.
2) PROPAGATION: Every node whose membership is *null* checks the membership of its neighbors one after one. As soon as it encounters a neighbor whose membership is not *null*, this node changes its own membership from *null* to the membership of that neighbor.
3) LOOP: Repeat STEP 2 until the fraction of nodes having *null* membership is below a preset threshold δ, or the number of times the propagation step has been invoked reaches a preset upperbound U.
4) (optional) HIERARCHY CONSTRUCTION: Recursively merging the clusters based on their closeness on the reference graph into a hierarchy.
5) LAYOUT: Finally, arrange nodes according to the resulting clusters. Nodes in the same cluster are laid out nearby in memory. If a cluster hierarchy is created, the leaf clusters are processed following their appearance order in the hierarchy.

The propagation step dominates the time cost of the algorithm. But it is a completely data-parallel process, meeting the strength of throughput-oriented processors.

Parameters and Adaptive Control: There are two parameters in the TLayout algorithm. The use of a small positive value of the parameter $delta$ allows the algorithm to stop with a small portion of nodes carrying *null* membership. These nodes will be attached to the end of the final data layout. As the number is small, they have little influence on the quality of the resulting data layout. But using such a value may save one or multiple invocations of the propagation step. Like many parameters used in practical systems, users set this value based on their experiences and preferences. We use 1% as its value for all our experiments.

The second parameter is the number of clusters K. A large value of K leads to quick membership propagation, hence few invocations of the propagation step. However, it may hurt the quality of the resulting data layout: Many nodes that have good reference affinity may fall into different clusters. The optimal value of K depends on the graph properties and the application. As the reference graph periodically changes throughout the execution of a dynamic simulation program, its value is difficulty for users to set.

We design an adaptive control to automatically adjust the value of K. After each data transformation, TLayout compares the transformation time and the length of the update period of the reference clue. If the transformation time is too long, TLayout doubles the value of K to accelerate the next data transformation. Typically, K starts with a small value (100 in all our experiments).

Implementation on GPU: TLayout is designed for general massively parallel architecture. We implement it using CUDA [2] in machines equipped with GPU. CUDA is a C-like interface for GPU programming. A CUDA program consists of a CPU code and a GPU code. The code executed on GPU is wrapped in functions called GPU kernels. A GPU typically contains hundreds of cores. There is a certain amount of on-chip memory (called shared memory) and a large chunk of off-chip memory (called global memory). When a GPU kernel is invoked, hundreds of GPU threads are launched to run the same GPU kernel with the same parameters. Each thread has one unique ID number; the kernel may use thread ID to trigger different behaviors of different threads.

In our implementation, each GPU thread manages one node in the graph. Algorithms 1 and 2 outline the CPU code and GPU kernel respectively.

Algorithm 1 *TLayout(num_nodes, num_edges, neighbor_list)*

```
1: // build a single array to store neighborhood info to prepare for GPU kernel execution
2: for $i = 0$ to $num\_edges - 1$ do
3:     $left = neighbor\_list[i][0]$;
4:     $right = neighbor\_list[i][1]$;
5:     if $neighbor\_size[left] < MAX\_NB\_Per\_Node$ then
6:         $neighbors[neighbor\_size[left]++] = right$;
7:     end if
8:     if $neighbor\_size[right] < MAX\_NB\_Per\_Node$ then
9:         $neighbors[neighbor\_size[right]++] = left$;
10:    end if
11: end for
12: //randomly select K nodes as the seeds for K clusters
13: for $i = 0$ to $K - 1$ do
14:     $membership[rand()\%(num\_nodes)] = i$;
15: end for
16: while too many nodes have null membership do
17:     //invoke GPU kernel for neighborhood-based clustering
18:     TLayoutKernel<<< ... >>>(neighbors, membership);
19: end while
20: //merge clustering results to generate a new data order
21: for $i = 0$ to $num\_nodes - 1$ do
22:     $id\_cluster = membership[i]$;
23:     $cluster\_lists[id\_cluster].append(i)$;
24: end for
25: $ind = merge\_lists(cluster\_lists)$;
26: return $ind$
```

Discussions: TLayout has some appealing characteristics worth mentioning. First, it well exploits the massive parallelism of GPU. Assigning a thread to every node makes the algorithm simple to implement and proceed efficiently. Second, a node having a high degree tends to grab more nodes into its cluster than other nodes do, which is a desirable property for spatial locality. Third, the algorithm adaptively selects the appropriate number of clusters. This adaptivity

Algorithm 2 *TLayoutKernel(neighbors, membership)*

```
1: // load neighbors into shared memory
2: . . .
3: // membership propagation
4: $i = global\_thread\_number$;
5: for $j = 0$ to $MAX\_NB\_Per\_Node - 1$ do
6:     $neighbor = get\_neighbor()$;
7:     if $membership[neighbor]$ & $!membership[i]$ then
8:         //propagate membership
9:         $membership[i] = membership[neighbor]$;
10:        break;
11:    end if
12: end for
```

fits the dynamic properties of irregular simulation well. We note that TLayout specifically exploits the massive parallelism in throughput-oriented devices (e.g., GPU). It is not intended to be used on CPU. (Experiments show it is tens of times slower than RCB on a CPU.)

V. Asynchronous Data Transformation Library (ATrans)

We integrate the techniques, along with previous transformation techniques, into a Asynchronous Data Layout Transformation library (ATrans) to simplify their use. ATrans consists of all the support for asynchronous transformation, the adaptive TLayout algorithm, and a set of previously implemented data transformation functions from University of Maryland [12]. It supports the asynchronous data transformation on both CPU and GPU.

Its usage is simple. To enable asynchronous data transformation for an application, it typically requires just an insertion of three function calls in the application program, one in the initialization stage, one after the update of the reference clue, and one at the beginning of the central loop (e.g., the time-step loop in Moldyn). Figure 6 illustrates the use of the library for Moldyn. The *ATrans_init_pipeline* function indicates whether CPU or GPU is to be used for analysis component, creates a helper thread, initializes the state of the pipeline and necessary data structures, and prepares the GPU execution if GPU is used. When the interaction list is updated, the *ATrans_analysis* function checks the pipeline state and wakes helper thread up to do transformation analysis if necessary. At the beginning of each iteration of the time-step loop, the *ATrans_reposition* function checks the state and reposition the data if it is time to do so.

```
int main (int argc, char **argv)
{
  /***initialization of the simulation***/
  ATrans_init_pipeline(__ATRANS_CPU, interaction_list,
    MAX_EDGE, coordinates, MAX_NODE, forces);
  for(iter = 0; iter < NUM_ITER; iter++)
  {
   if(update_interaction_list() == true)
     ATrans_analysis();
   ATrans_reposition();
   /***simulation kernel code***/
  }
  /***deal with result***/
}
```

Fig. 6. Use of the ATrans library in Moldyn. Inserted codes are function calls with prefix "ATrans_".

VI. Adapting On The Fly

The benefits of asynchronous transformation do not come for free. Recall that to circumvent the data dependence, it uses obsolete reference clues as heuristics for data transformations. Although in many cases the benefits outweigh the catch, it is not always so. Whether an asynchronous transformation excels a synchronous transformation is subject to the ratio between transformation overhead and per iteration computation time, the frequency of the update to reference clues, the speedup, and so on.

We devise an online adaptive scheme to select the suitable transformation strategy on the fly. The basic idea is to estimate the benefits of different strategies during the initial time steps, and then apply that strategy to the remaining time steps.

To figure out the overall benefits of synchronous transformation, it is necessary to determine the best frequency to apply it. Because of its transformation overhead, applying it at every update of reference clues is often sub-optimal. We employ a prior method [20] to solve the problem. By applying the synchronous transformation only once, it can determine the best frequency and estimate the overall benefits by observing the computation speed in a number of iterations following the transformation. The process introduces no extra overhead.

Figuring out the overall benefits of asynchronous transformation is less straightforward. As asynchronous transformation is off the critical path, it can be applied often. In our design, it is applied at the reference clue update following the finish of the previous asynchronous transformation. Because the per iteration time varies across update periods, it is difficult to get a closed form to compute all the ending/starting time points of asynchronous transformations, causing difficulty for benefit estimation.

Our solution is to emulate the timeline of the kernel computation and the applications of asynchronous transformations. For space constraint, we describe it briefly. It requires the following parameters: the analysis and reposition times of a data transformation, the frequency of reference clue update, the total time steps of the kernel computation, and computation speed in a number of iterations following the transformation. Attainment of these numbers needs one application of the transformation only. An emulation of the timeline involves the computation of a number of linear expressions for calculating when each asynchronous transformation will apply and how many time steps of computation can benefit from it. The emulation takes less than 0.01% of overall running times in all our experiments.

After estimation of the overall benefits of synchronous and asynchronous transformations, the winner will serve for the rest of the execution. Although the program may not be using the optimal transformation scheme during the initial time steps, the next section will show that the influence is small as these steps take only a small portion of the entire simulation.

The adaptive scheme may suffer if some key factors (e.g., frequency of reference clue update) change dramatically across time steps. Fortunately, most dynamic simulations do not see such drastic changes.

VII. Evaluation

We conduct a series of comparisons to evaluate the values of the techniques. We give an overview of the results first.

- *Asynchronous Transformation:* As Section III mentions, asynchronous transformation hides most overhead, but its use of obsolete reference clues may have certain side effects on the resulting locality. We conduct a head-to-head comparison between asynchronous and synchronous transformations (both using RCB on CPU) in terms of the overall performance and resulting locality of the transformed applications. The result shows both the strength and weakness of the asynchronous transformations. On three benchmarks, asynchronous transformations lead to

18% more speedup than synchronous transformations do. In addition, for these benchmarks, using an extra CPU core for transformation brings 15% more speedup than using that core for computation, justifying the resource usage of the asynchronous transformations. On the other hand, the negative effects of obsolete reference clues outweigh the benefits of asynchronous transformations on two other benchmarks, leading to slightly less speedup than the synchronous transformations do.

- *Runtime Adaptation:* The runtime adaption scheme is able to identify the best transformation strategy for all benchmarks. With low overhead, it helps exert the strength of both asynchronous and synchronous transformations.
- *TLayout:* By comparing with a prior sophisticated algorithm (RCB), we observe that in most cases, the TLayout algorithm produces data layout of similar quality as the prior algorithm does, but takes around one third time to run.
- *Overall:* When the techniques are applied together, they generate 1.3–3.1X speedup over the original performance of five benchmarks, outperforming the state-of-the-art data transformation techniques significantly.

The conclusions obtained are based on measured wall-clock times and confirmed by hardware performance counters results.

A. Methodology

Platform: All experiments happen on a dual-socket dual-core AMD Opteron 2216 machine in the National Center for Supercomputing Applications. The machine is equipped with an NVIDIA Tesla S1070 GPU with 16GB DDR3 memory. It consists of four Tesla T10 C1060 GPUs, with each containing 240 cores, organized in 30 streaming multiprocessors. We use only one of the GPUs in our experiment. The machine runs Linux 2.6.33. We use GCC 4.3.2 (with "-O3" flag) as the compiler and CUDA 3.0 as the GPU programming model. We employ libpfm4 [1] for collecting cache performance data.

Benchmarks: We concentrate our experiments on a dynamic simulation benchmark suite from Han and Tseng [12], and two other programs, Mesh and CFD, respectively from the Chaos group [7], [30] and the Fluid Dynamics community [5]. The suite from Han and Tseng consists of three representative programs, Nbf, Irreg, and Moldyn. They are all derived from real applications. Nbf is abstracted from GROMOS, a force field of molecular dynamics simulation; Irreg is the kernel of an iterative partial differential equation solver; Moldyn is from a molecular dynamics simulation named CHARMM. These three benchmarks have been commonly perceived to be representative, and have served as the *only* benchmarks in some influential data locality studies in dynamic simulations [12], [15], [26]. We add two more benchmarks to increase the coverage. Mesh is an unstructured mesh simulation. CFD is an unstructured grid finite volume solver for three-dimensional Euler equations for compressible flow. Similar to the extensions Han and Tseng made to Irreg [12], both Mesh and CFD are modified to accommodate dynamic changes in the underlying mesh or grid structures.

Table I lists the properties of the inputs used with these benchmarks. FOIL and AUTO are 3D meshes of a parafoil and GM Saturn automobile, respectively. MOL1 and MOL2 are small and large 3D molecule models originally obtained from MOLDYN application. The results on all inputs show similar performance trends. Due to space constraints, our discussion concentrates on the results on large inputs (AUTO and MOL2) for the severity of their locality issues.

The frequency of the update to reference clues affects the problem setting and the potential of runtime data transformations. We experiment three typical frequencies: one update in every 10, 20, or 30 iterations of the main computation loop of the applications.

TABLE I
INPUTS*

Name	# Nodes	# Edges	Description
FOIL	144649	1074393	3D mesh of a parafoil
AUTO	448695	3314611	3D mesh of GM's Saturn
MOL1	131072	1179648	3D molecule distribution (sm)
MOL2	442368	3981312	3D molecule distribution (lg)

*: come from Han and Tseng [12].

Transformation Frequency: Data transformation can be applied as often as once per update of the reference clue, or once every several updates. The more frequent it is applied, the better the locality of the application is, but meanwhile, the more overhead it incurs.

For the asynchronous paradigm, the transformation frequency is automatically determined by the 6-state master-helper coordination scheme as described in Section III-D. The problem is tricky for synchronous transformations. For a fair comparison, one seemingly straightforward option is to use the same transformation frequency as the asynchronous transformation uses. But this option is in fact unfair to the synchronous scheme. That frequency often causes much worse performance than some other frequencies for synchronous transformations. A previous study [20] introduces a method to analytically determine the optimal frequency for synchronous transformations. We have verified the optimality of the method through a sequence of empirical measurements. In all our experiments, we use optimal frequencies found in that way for synchronous transformations.

Algorithm and Others: We select previously proposed RCB algorithm (implemented by Han and Tseng [12]) as the analysis algorithm in all CPU experiments, synchronous or asynchronous. RCB has been shown to be one of the most sophisticated algorithms that produce the largest locality enhancement for most benchmarks [12]. Our experiments echo that despite it is more expensive than some other methods (e.g., CPACK), its overall performance is often among the best when it is applied synchronously at the best frequency.

As the asynchronous transformation is mainly on data reordering, we apply the same computation reordering (lexicographical sort) to all experiments. The computation reordering overhead is small (less than one seventh of RCB) and is counted in data repositioning overhead in all experiments.

B. Experimental Results

We experiment both single-thread and parallel executions of the benchmarks. They show similar conclusions. We first give a detailed analysis using the single-thread results, and then report the parallel results at the end, along with the justification of the resource usage by asynchronous transformations.

1) Sequential Executions: Figure 7 shows the comparison of overall running times. Each time consists of the application running time and all transformation overhead that is not hidden (including data transfer between CPU and GPU).

IRREG, NBF, MOLDYN: On the first three benchmarks, the synchronous transformations show 77% average speedup. The asynchronous transformations on CPU show 18% more average speedup. The benefits come from two aspects. First, the asynchronous scheme hides significant transformation overhead, as Figure 8 reports. The second benefit relates with the first. Because the transformation incurs smaller overhead on the critical path than the synchronous scheme does, it is automatically applied more frequently by the master-helper coordination scheme than the synchronous one. The

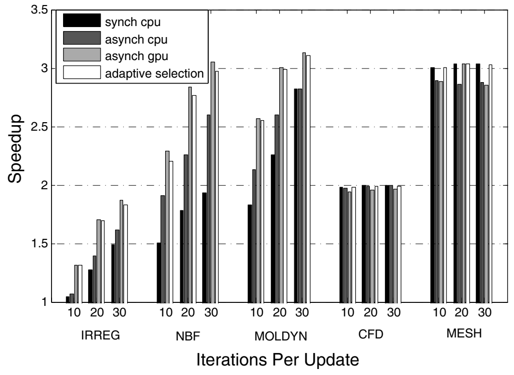


Fig. 7. Speedup of the overall executions for single-threaded benchmarks. The speedups are over single-threaded benchmarks without any data transformation applied.

more frequent transformation yields better locality, confirmed by the L2 cache miss rates shown in Figure 10. The figure shows a few exceptional cases (e.g., the configuration ”Moldyn 30”), in which, the two transformations are applied at the similar frequencies; the use of obsolete reference clues causes the relatively less locality enhancement. However, thanks to the overhead hiding by the asynchronous transformation, it leads to the better or similar overall performance still, as Figure 7 shows.

We stress that the synchronous results are what we get when the optimal transformation frequency is used. Increasing invocation frequency of the synchronous transformations yields only worse overall performance due to the large overhead incurred, while decreasing the frequency worsens the performance as well due to the less locality enhancement to the application, as Figure 9 illustrated.

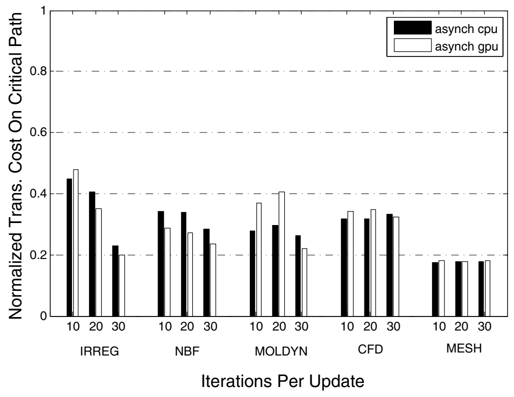


Fig. 8. Optimization cost on critical path. The results are normalized over those of synchronous transformation.

The asynchronous TLayout produces even larger benefits than the asynchronous CPU approach. The extra speedup ranges from 28% to 112% with an average of 65% over those of the synchronous scheme, and 25–58% better than those of the asynchronous CPU results. The extra benefits come from two appealing features of the TLayout algorithm. First, it runs 2.8 to 3.3 times faster than the RCB algorithm, thanks to its effective exploitation of the throughput-oriented processors. Second, it produces data layout of comparable quality as the sophisticated RCB algorithm does as Figure 11 reports. These two features together explain why asynchronous TLayout

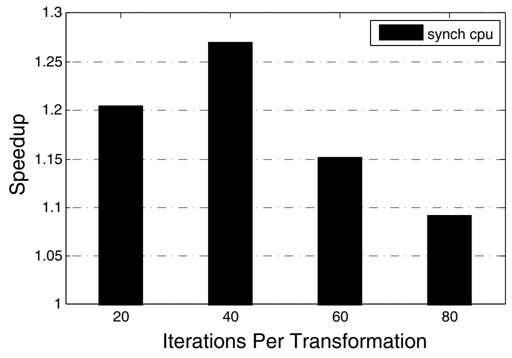


Fig. 9. Speedup of IRREG with different transformation frequencies. Neighbor list is updated every 20 iterations.

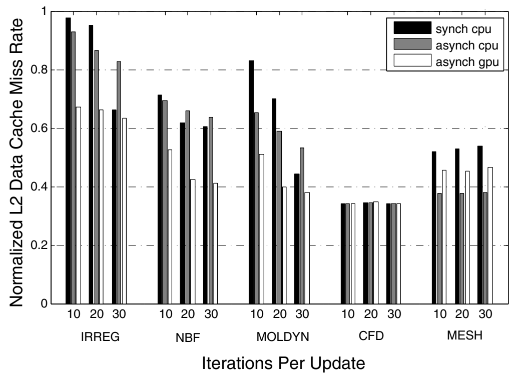


Fig. 10. L2 cache performance comparison between synchronous and asynchronous data transformation. Results are normalized over those without any transformation.

produces better data locality than the asynchronous CPU does. The second feature ensures that each invocation of the data transformation in the two schemes are similarly powerful, while the first feature entails much more affordable invocations of data transformations in the asynchronous TLayout than in the asynchronous CPU.

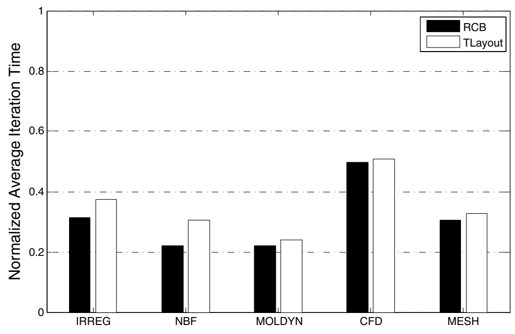


Fig. 11. The time per iteration of the computation loop after a transformation is applied. It is the average of 100 iterations following the transformation. The results are normalized over those of single-threaded benchmarks with no transformations applied.

CFD and MESH: The results on these two programs show a trend different from the other three programs. On both of them, changes to the reference clue during the simulations are less significant than on the other programs. The overall speedups from the transformations are still large because the initial data layout is inferior. However, because the changes are small during the simulation, there is no need to apply transformations often. There is limited overhead for asynchronous transformations to hide. Consequently, the negative effects of the use of obsolete reference clues become noticeable. So on both programs, regardless the reference clue update frequencies, the asynchronous transformations perform slightly worse than the synchronous transformation. The L2 cache results of Mesh in Figure 10 seem counter-intuitive: Asynchronous ones are lower than the synchronous one. A plausible reason is that the locality of the program is mainly embodied by other metrics. For instance, the synchronous scheme has L1 cache miss rate half of that of the asynchronous GPU scheme.

Adaptive Selection: The adaptive selection scheme successfully selects the best strategy to use for all cases. Because some transformations in the initial time steps do not use the optimal strategy, there are slight differences between the speedups from the adaptive scheme and those of the best strategy. However, overall, it achieves the near best performance on all benchmarks, showing the promise for exerting the strength of both asynchronous and synchronous transformations.

2) Parallel Executions: Figure 12 reports the similar comparison but on parallel executions of the benchmarks. For the baseline (i.e. no transformations applied) and "synchronous CPU", we use 4 threads for each benchmark as the machine contains 4 cores. In "asynchronous CPU" and "asynchronous GPU" case, we use 3 threads for each benchmark so that the transformation can happen on the remaining core.

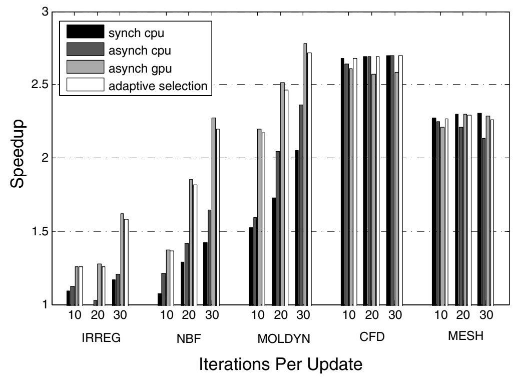


Fig. 12. Speedup of the overall executions for parallelized benchmarks. The speedups are over parallelized benchmarks without any transformation.

The results show similar conclusions as the single-thread experiments do. One particular point we want to mention is that even though the "asynchronous CPU" uses one fewer worker threads than "synchronous CPU", with the help from the asynchronous transformation, it still excels in resulting performance. Part of the reason is that the irregular applications have many communications among threads due to the inherent properties of the applications. As a result, the parallel program shows sub-linear performance scalability in the number of threads. Adding the fourth worker thread improves the performance of the programs by 6%, exceeded by the benefits from the asynchronous transformations. The results justify the resource usage of the asynchronous transformations. In addition, the parallelization imposes different influence on the locality of different benchmarks. The locality issue of CFD becomes especially serious after the parallelization, hence the large benefits from data

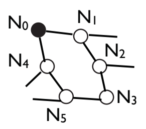


Fig. 13. An example showing the membership propagation in TLayout. The filled node is already clustered; the others are not.

transformations.

3) More Results on TLayout Algorithm: The speed of membership propagation in the algorithm determines the number of iterations the propagation has to happen (to reach the predefined threshold δ). In all the experiments reported in previous sub-sections, the average numbers of needed propagation iterations are no larger than four. This result indicates the high speed of membership propagation. Analytically, it may be attempting to think that if the closest center is K hops away from a node, it would take K iterations of propagation for that node to be clustered. However, because global memory is used for membership labels, during an iteration of propagation, the membership of a node becomes visible to all threads (e.g., all nodes) immediately after the node gains its membership. For instance, in Figure 13, node N_3 can be clustered in one propagation if either of the following two conditions is met: (1) N_3 is visited after N_2 and N_2 is visited after N_1; (2) N_3 is visited after N_5 and N_5 is visited after N_4. In TLayout, the visiting order of nodes is random; in the GPU implementation, the order is determined by the scheduling of GPU threads, which exhibits large randomness.

To examine the scalability of the algorithm, we create a spectrum of problems of different sizes. At each size, we run the algorithm 7 times to get the average number of propagations required to cluster 99% nodes. As the focus is on assessing the propagation speed, we fix the number of clusters to be 100 in all runs. Results in Figure 14 demonstrates the good scalability of the algorithm.

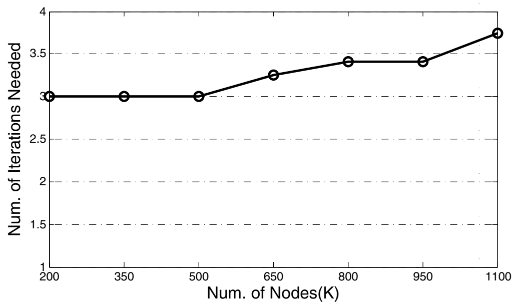


Fig. 14. Scalability of TLayout

Overall, the results demonstrate that both the asynchronous data transformation and the TLayout algorithm are able to produce certain degrees of benefits for the enhancement of data locality of irregular dynamic simulations. Together with the online adaption scheme, they resolve the quality-dilemma faced by existing data transformation techniques, and yield significant performance improvement.

VIII. Related Work

In Section II, we reviewed some prior data reordering and computation reordering techniques for the enhancement of data locality of dynamic simulation programs. In addition, Strout and others have proposed a compile-time framework that allows the explicit composition of run-time data and iteration reordering transformations [26]. Kulkarni and others [16] have studied locality issues of irregular applications in the context of optimistic parallelism. They concentrate on the partition of data among threads rather than data layout reorganizations for locality improvement.

Recent years have seen a rapid increase of the use of GPU for data-parallel computing. Previous work on CPU-GPU cooperative computing concentrates on offloading some computation-intensive and easily parallelizable parts of an application to GPU. In this scenario, the key issue is how to partition the jobs among GPU and CPU [24], and how to optimize GPU code to maximize the computing efficiency on GPU through compiler techniques [3], [17], [27], runtime optimizations [28], [29], or empirical search-based optimizations [19]. Some recent studies attemp to enable seamless translation between GPU and CPU code [9], [10], [25]. We are not aware of prior proposals in using GPU to do runtime optimizations for CPU computing.

There are many clustering algorithms developed in the machine learning area [13]. But most of them are distance-based (e.g. K-Means) rather than topology-based. Our search yields no satisfied topology-based clustering algorithm that is simple and fits GPU well, hence our development of TLayout.

IX. Conclusion

This paper presents three techniques for resolving the quality-overhead dilemma of data transformations for irregular references. The first, *asynchronous data transformation*, moves data reordering off the critical path through dependence circumvention and layout approximation. The second, *TLayout*, is a novel data transformation algorithm designed to take advantage of modern throughput-oriented processors. The third technique, *adaptive control*, allows transparent selection of suitable transformation schemes for an execution. Together, they improve the performance of some irregular dynamic simulations significantly. In addition, this study initiates a new way of collaborations between CPU and co-processors, which may lead to some unconventional directions for program optimizations in a heterogeneous computing environment.

Acknowledgement

We owe the anonymous reviewers our gratitude for their helpful suggestions on the paper. We thank Michelle M. Strout for her comments and NCSA at UIUC for devices. This material is based upon work supported by the National Science Foundation under Grant No. 0811791 and 0954015 and IBM CAS Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or IBM.

References

[1] libpfm4. http://perfmon2.sourceforge.net/docs.html.

[2] NVIDIA CUDA. http://www.nvidia.com/cuda.

[3] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proceedings of ICS*, 2008.

[4] M. Berger and S. Bokhari. A partitioning strategy for non-uniform problems on multiprocessors. *IEEE Trans. Computers*, 37(12):570–580, 1987.

[5] A. Corrigan, F. Camelli, R. Lohner, and J. Wallin. Running unstructured grid based cfd solvers on modern graphics hardware. In *Proceedings of the 19th AIAA Computational Fluid Dynamics*, 2009.

[6] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured euler solver using software primitives. In *Proceedings of the 30th Aerospace Science Meeting*, 1992.

[7] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributioned memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, 1994.

[8] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64(1):108–134, 2004.

[9] Z. Guo and X. Shen. Fine-grained treatment to synchronizations in gpu-to-cpu translation. In *Proc. of the Workshop on Languages and Compilers for Parallel Computing*, 2011.

[10] Z. Guo, E. Zhang, and X. Shen. Correctly treating synchronizations in compiling fine-grained spmd-threaded programs for cpu. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2011.

[11] H. Han and C. W. Tseng. Improving locality for adaptive irregular scientific codes. In *Proceedings of Workshop on Languages and Compilers for High-Performance Computing (LCPC'00)*, White Plains, NY, August 2000.

[12] H. Han and C.-W. Tseng. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel Distributed Systems*, 17(7):606–618, 2006.

[13] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer, 2001.

[14] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. In *Proceedings of ICPP*, 1995.

[15] S. Kim, H. Han, and K. Choe. Region-based parallelization of irregular reductions onexplicitly managed memory hierarchies. *Journal of Supercomputing*, 2009.

[16] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. In *Proceedings of ASPLOS*, pages 233–243, 2008.

[17] S. Lee, S. Min, and R. Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *Proceedings of PPoPP*, 2009.

[18] W. Liu and A. Sherman. Comparative analysis of the cuthill-mckee and the reverse cuthill-mckee ordering algorithms for sparse matrices. *SIAM J. Numerical Analysis*, 13(2), April 1976.

[19] Y. Liu, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for gpu programs optimization. In *Proceedings of International Parallel and Distribute Processing Symposium (IPDPS)*, pages 1–10, 2009.

[20] G. Marin, G. Jin, and J. Mellor-Crummey. Managing locality in grand challenge applications: a case study of the gyrokinetic toroidal code. 2008.

[21] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of SC*, 1999.

[22] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proceedings of PACT*, 1999.

[23] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *Proceedings of ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 2002.

[24] V. Ravi, W. Ma, D. Chiu, and G. Agrawal. compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceedings of ICS*, 2010.

[25] J. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W. Hwu. Efficient compilation of fine-grained spmd-threaded programs for multicore cpus. In *CGO '10: Proceedings of the International Symposium on Code Generation and Optimization*, 2010.

[26] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[27] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *PLDI*, 2010.

[28] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 369–380, 2011.

[29] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen. Streamlining gpu applications on the fly. In *Proceedings of ICS*, 2010.

[30] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 255–266, June 2004.