Improving Inter-thread Data Sharing with GPU Caches

Abstract

The massive amount of fine-grained parallelism exposed by a GPU program makes it difficult to exploit shared cache benefits even there is good program locality. The non deterministic feature of thread execution in the bulk synchronize parallel (BSP) model makes the situation even worse.

Most prior work in exploiting GPU cache sharing focuses on regular applications that have linear memory access indices. In this paper, we formulate a generic workload partitioning model that systematically exploits the complexity and approximation bound for optimal cache sharing among GPU threads. Our exploration in this paper demonstrates that it is possible to utilize GPU cache efficiently without significant programming overhead or ad-hoc application-specific implementation.

1. Introduction

Modern GPU programs use fine grained parallelism in which a large task is partitioned into many small ones and each thread handles its own part. In a lot of applications, large amount of data is shared and reused not only by different instructions within a single thread, but also between different threads. Data sharing in GPU programs gives rise to memory performance enhancement opportunities but these data sharing opportunities are challenging to be exploited in GPU.

Exploring the data sharing with cache in GPU programs is challenging for several reasons. First of all, GPU programs are highly parallel and concurrent. The non-deterministic thread interleaving for 10^5 threads or more makes it extremely easy to pollute the cache. Secondly, programmers do not have enough control in hardware scheduling strategies. Thirdly, unlike CPU programs, using control-flow code to check if data is also used in other threads, is expensive in GPU programs due to the inefficiency of single instruction multi-thread (SIMT) architecture in handling dynamic control divergences.

Least but not last, even if all the aforementioned difficulties in control, scheduling and thread interleaving are solved, there remains this fundamental question, how can we map workload into large number of thread block(s) so that data is maximally shared within the same thread block(s) and data communication is minimized across thread blocks.

We show the thread interaction pattern of a real application – computational fluid dynamics application (CFD) in Fig. 1. We use data input from Rodinia benchmark suite [Che et al. 2009] and plot the full interaction for first twenty threads. Every node represents a particle and every edge represents interaction between two particles. Every particle is handled by a thread, which will calculate its interaction with all neighbour particle.

There are two things worth noticing in the graph of CFD. Firstly, even though every node has degree ≤ 4, the graph is complicated enough for us to find the optimal partition of particles or threads. Secondly and more interestingly, we mark every node with its associated thread number, in Fig. 1, it can be seen that threads that are close to each (running at the same time), for instance threads 1 to 8 are assigned to particles that are not near each other.

Prior work in inter-thread data sharing in GPU program are either focused on regular memory-access pattern or spatial locality (memory coalescing). The polyhedral compilation framework is used to optimize data locality in automatic parallelization of affine loop nest [Baskaran et al. 2008] [Bondhugula et al. 2008]. These techniques work well for applications with regular memory accesses pattern well, but not for other irregular applications. Regarding irregular GPU programs, memory access coalescing has been studied extensively [Yang et al. 2010] [Liu et al. 2013] [Wu et al. 2013]. Data coalescing helps enhance spatial locality by grouping data objects used by co-running threads together, but it does not necessarily handle data reuses and sharing across threads. In CPU programs, sequential or with moderate parallelism, code transformation and dynamic techniques such as inspector/executor [Strout et al. 2003] based reordering and data packing [Ding and Kennedy 1999] have

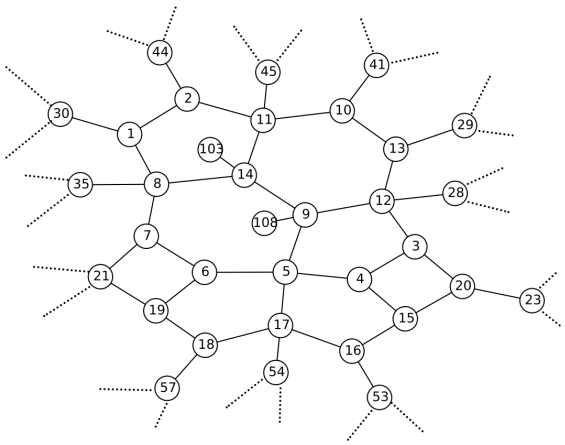

Figure 1. Motivation Application: Computational Fluid Dynamic (CF) Simulation

been well applied. There is a lack of systematic study for data reuse in irregular applications running massively parallel GPU architecture.

In this paper, we focus on addressing the problem of workload partitioning for maximal GPU cache performance. We propose a model called *data-affinity graph edge partition model* which accurately characterizes GPU cache performance. We rigorously obtain the time complexity of and the bound of approximation algorithm for this model. This problem is NP-complete problem, however, we are able to bound approximation algorithm to a fixed constant related to node degree and the number of partitions. We are also able to design a fast and effective polynomial-time partition approach that yields good performance improvement of various benchmarks and input sets.

We summarize our contributions as follows:

- Our work in this paper reveals the time complexity of GPU computation partition problem with respect to cache performance as NP-hard.
- Despite the fact this is a NP-hard problem, we proposed a two-stage approximation solution with bounds. As far as we know, this is the first time an approximation approach with bounds is obtained for GPU cache sharing problem. With approximation approach, we gain insights on how far we are away from optimal solution in different scenarios, for instance, different graphs and different degree distribution.
- The solution to the workload partition problem in the data-affinity graph model also effectively guides the strategy of data layout placement.
- In addition to bounded approximation approach, we also provide an practical solution framework and a runtime library GRAPE (GRAph Partition on Edge) that includes efficient heuristics augmented to the approximation solution and handling of other practical issues including software cache pre-loading and index maintaining.

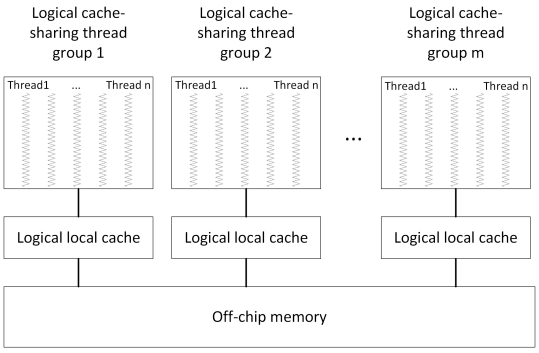

Figure 2. Logical threads cache sharing model

2. GPU Cache Sharing Model

GPU uses an uniform cache sharing model. A GPU consists of multiple streaming processors (SM). Every SM has local cache of the same size. The streaming cores on one SM share cache while cores across different SMs do not.

There are two types of cache in every SM: software-managed cache and hardware-managed cache. The scope of sharing for software and hardware cache is different. For software cache, data sharing occurs within a thread block [1]. For hardware cache, data sharing occurs among multiple thread blocks running on the same SM. We refer to the thread block(s) that share cache as a cache-sharing thread group.

We show how logical threads share cache in Fig. 2. Every logical cache-sharing thread group corresponds to a logical cache. Note that in GPU execution model, typically there are more logical thread groups than physical local caches or SMs. A set of thread blocks *occupy* one SM until they finish all their work and another set of thread blocks occupy the SM again, called occupancy in NVIDIA terminology. For this reason, we treat it as if there is a fixed-size logical cache for every logical thread group.

A data object loaded into one logical cache can be reused by threads in the same cache-sharing thread group. A data object that is used by different cache-sharing threads groups need to be loaded into multiple caches. The minimal number of loads is equivalent to the number of data objects involved in computation, meaning every data object has to be loaded at least once from off-chip memory.

3. Complexity, Bounds and Approximation

To effectively use shared cache in GPU programs, we need to maximize data reuse within a cache-sharing thread group and minimize communication across cache-sharing thread groups. In GPU programming model, every thread executes the same piece of code on different inputs, we map a set of

[1] We use NVIDIA terminology throughout this paper

computations of the same type into different thread groups. The workload assignment process determines how much and how often data is shared within cache-sharing thread groups.

3.1 Data-affinity Graph Edge Partition

We propose **data-affinity graph model** and our model precisely characterizes the number of memory loads of any workload mapping strategy. In our proposed data affinity graph, we use edges to represent computation and perform balanced edge partitioning for workload assignment.

We define a data-affinity graph $D = (V, E)$ with the set of vertices V and the set of undirected edges $E \subset V \times V$. Let n and m denote the number of vertices and the number of edges, respectively. A vertex $v \in V$ represents a data object. An edge $e \in E$ denote a computation operation that involves the two data objects, for instance, the force calculation between two particles in molecular dynamics simulation.

Assume every data object is of the same size and every edge denotes computation of the same type. An edge partition of edges into k clusters so that every edge is assigned to exactly one cluster. Every cluster corresponds to a cache-sharing thread group defined in Section 2.

Let p_v denote the number of clusters in which vertex $v \in V$ appears. A vertex v appears in a cluster i if any of its incident edges is assigned to cluster i. We define a communication cost C which has to be minimized under a condition of balanced loads of the cluster. Let $C = \sum_{v \in V}(p_v - 1)$ and L_i denote the load of cluster i, our *k−way balanced edge partition* problem can be written as:

$$\begin{array}{ll} \min & C(x) \\ \text{s.t.} & \forall i \in [k]\ \ L_i(x) = \frac{m}{k} \\ & x \textit{ is a valid edge partitioning} \end{array} \quad (1)$$

The communication cost C defined here is the extra number of memory loads beyond the original one load of every data object. It is because a data object loaded into a local cache can be visible to the whole data-sharing thread group.

We call the set of nodes that appear in two or more clusters split set in order to emphasize the difference between our notion and the similar notion of "cut set" in graph theory.

3.2 Complexity and Optimality

It is easy to prove, by a reduction from the NP-hard **partition problem**, that the *k-way* balanced edge partition problem is NP-hard, even for $k = 2$.

We briefly sketch the reduction. Recall that in the partition problem we are given a set S of positive integers $a_1, a_2, ..., a_n$, and our task is to decide if the numbers can be partitioned into two subsets S_1 and S_2 such that the sum of numbers in S_1 equals the sum of the numbers in S_2.

In the reduction we construct a graph with n *cycles* $c_1, c_2, ..., c_n$ such that $|C_i| = a_i$. Obviously, a partition (S_1, S_2) for the instance of the partition problem exists if and only if the $2-$way balanced edge partition problem reduce from it has a minimal communication cost of 1.

3.3 Approximation Algorithm and Analytical Bound

Although the k-way balanced edge partition problem is hard in the worst case, we have designed an approximation algorithm that works very well in many cases, and with provable guarantees. In practice, the basic algorithm can be auxiliary with powerful heuristic approaches, yielding overall good performance. Our algorithm, in particular, performs favorably for the important class of irregular applications in dynamic scientific simulation.

The basic idea of our approach is to map the edge partition problem into a vertex partition problem. We then use the solution of the vertex partition problem to reconstruct the solution to the original edge partition problem.

Two-Stage Data-affinity Graph Edge Partition

Stage 1: Mapping and Partitioning

We map the original data affinity graph $D = (V, E)$ into another graph $D' = (V', E')$ such that for every node $v \in V$ of degreed d, there are d corresponding cloned nodes $v'_1, \ldots v'_d \in V'$. We add edges to connect these d nodes to form a *cycle* and the order of the d nodes in the cycle is arbitrary. We call these edges **auxiliary**, and assign weight half to them.

For every edge $e \in E$ with end points $u, v \in V$, there is a corresponding edge $e' \in E'$ with end points $u'_i, v'_j \in V'$ with the restriction that the image of two different edges never share an end point in D' (it is easy to see that this is doable, since a node v in D is split into $\deg d$ nodes in D'). We call the images of the original edges **real**. Real edges are assigned weight one.

The number of vertices in D' is exactly twice the number of edges in D since every real edge is associated with two vertices that are not incident to any other real edge. We now perform a vertex partition algorithm for the graph D', i.e. a one that decomposes the vertex set V' of D' into k equal parts while minimizing the total number of edges that are incident to two different parts, also called **cut edges**. There are two different scenarios: If all cut edges are auxiliary edges, the solution naturally maps back to a balanced edge partition solution of D.

Otherwise, we perform the steps described in stage 2 in order to turn the solution into a one in which only auxiliary edges remain in the cut set.

Stage 2: Edge Fixing

If the cut set of D' contains a real edge, one end point of the this edge needs to be moved to the cluster of the other end point.

Since the above manipulation will upset equality between clusters, in order to minimize (and eventually eliminate) the harm caused by the moves, we process real edges that cross between different clusters in the following order.

We cycle through all pairs (A', B') of clusters and do as follows: For the current pair (A', B') of clusters we first find the set $R_{A',B'}$ of real edges that cross between A' and B'. Then we put half of the edges of $R_{A',B'}$ entirely into A' and the other half entirely into B'. If $|R_{A',B'}|$ is odd, we choose a single edge from it and leave its end points in the same clusters as they were originally (i.e. in A' and B'). It is easy to see that restructuring the clusters as above leaves them balanced.

Consider now every cluster as a "super-node," and form an edge between two super-nodes if after the restructuring there is an edge between the corresponding clusters. A graph will arises on the super-nodes that we call C'.

For technical reasons we shall assume that the cluster size, $2m/k$, is even (this is a natural assumption, since if k does not divide m, the original problem cannot be solved). We argue that in this case the degree of every super-node in C' is even: Take out the inner-cluster edges together with their end-points in D'., and all we are left with are nodes of internal real edges (that are not cut), with an even number of end points. Thus the number of edges going out of a cluster must be even too.

Since C' has only even degree vertices, its edge set decomposes into Euler-tours that can be made directed in an obvious way. In a final stage we can fix edges along these directed Euler tours by restructuring the clusters once more, at this time putting each inter-cluster edge into the cluster associated with its starting point in the directed Euler tour. It is easy to see that after this final restructuring of clusters they remain of equal size.

Edge-fixing process can be designed in a more sophisticated way rather than arbitrarily picking pairs of clusters and cut real edges, which we will describe in Section 4.

Analytical Bounds

Let d_{max} be the maximal degree in graph D. We can guarantee an edge partition solution for D that is within a factor of $\frac{1}{2}d_{max}$ optimal if the vertex partition algorithm we use for for D' is optimal. (If it is not, we loose an additional factor, correspondingly).

Recall that we have called the value $C = \sum_{v \in V}(p_v - 1)$ in Section 3.1 communication cost. (This is actually the cost over n. Cost $|V| = n$ *must* incur even in the best case.) We denote the optimal communication cost by $optC$.

The result above provides a good bound for a large class of dynamic scientific simulation applications that perform simulation in two-dimensional or three-dimensional grid such as computational fluid dynamics (mentioned as motivation application), molecular simulation and quantum physics simulation.

We further present a refined practical approach that bounds the communication cost to $T \times d_{avg} \times optC + \epsilon n(k-1)$ (assuming a vertex partitioning algorithm that achieves an approximation factor of T is available) for graphs that have large max-degree nodes and small average degree. This refined approach works well for data-affinity graph that has scale-free degree distribution. In our practical solution framework described in Section 4.

Proof

In the k-way balanced edge partition, the optimal split for data-affinity graph D can be transformed into a cut for balanced vertex partition in D'. We simply need to cut every cycle in D' that corresponds to every node in Ds split set $v \in split(D)$.

In the worst case, every edge in the cycle needs to be cut, thus the cut set size is $\sum_{v \in split(D)} d_v/2$. Thus the optimal cut set size in D' should not be greater than $\sum_{v \in split(D)} d_v/2$ since such a cut weight exists.

A balanced vertex partition in D' can be mapped to a balanced edge partition in D using edge-fixing steps described by our algorithm. Any real edge (of weight 1) in D' cut set incurs 1 extra load when we move one of its end-point in the edge-fixing algorithm. An auxiliary edge of weight $\frac{1}{2}$ in a cut set of D' costs at most 1 extra load.

Obviously, the auxiliary edge cuts in a cycle need to come in pairs. An auxiliary edge may be used by two adjacent pairs of cuts (for three different clusters) and only counted as half unit weight. Therefore, an auxiliary edge of weight 0.5 might contribute to one extra load. Note that for a 2-way cut, an auxiliary edge of weight 1/2 contributes 0.5 extra memory load since there are only 2 different clusters.

Henceforth a cut set of D with total edge weight M leads to an edge cut solution that gives at most $2M$ extra loads as communication cost $C = 2M$. Since M is bounded by $\sum_{v \in split(D)} d_v/2$, $C = 2M \leq \sum_{v \in split(D)} d_v$. The actual optimal communication cost is $\sum_{v \in split(D)}(p_v - 1)$. We can get the ratio between C and actual optimal:

$$\begin{aligned} \frac{C}{optC} &\leq \frac{\sum_{v \in split(D)} d_v}{\sum_{v \in split(D)}(p_v - 1)} \\ &\leq \frac{\sum_{v \in split(D)} d_v}{|split(D)|} \\ &\leq average(d_v), v \in split(D) \\ &\leq d_{max} \end{aligned} \tag{2}$$

For $k = 2$, since we do not need to multiply M by 2, the bound of $\frac{C}{optC}$ becomes $\frac{d_{max}}{2}$.

As shown in the derivation process of inequality (2), the ratio to the actual optimal depends on multiple factors such as average degree of node in optimal split set, average clusters the split set node v falls into p_v. But at worst it is d_{max} or $d_{max}/2$.

We use optimal vertex partition solution to reconstruct an edge partition solution. The vertex partition problem is also a NP-hard problem. However, it has been studied intensively in past few decades. There is a fast and efficient multi-level vertex partition algorithms. We use the METIS [Karypis and Kumar 1995] library to perform vertex partition. If the obtained vertex partition solution is close to optimal, then our reconstructed edge partition can also get closer to optimal.

For data affinity graphs that have large maximal degree and maximal degree is much larger than average degree, for instance, scale-free data-affinity graph for sparse matrix vector multiplication (IMDB for instance), we use a different approach to bound the worst case scenario using average degree. The approximation bound is:

$$C \quad \leq \quad T \times d_{avg} \times optC + \epsilon n(k-1) \qquad (3)$$

This category of graphs for which $d_{max} >> d_{avg}$ typically has a small percentage of nodes that have very large degrees, including scale-free graphs whose node degree distribution follow power law. We treat them differently than other nodes since they may prevent us from finding good edge partition based on the d_{max} factor result in inequality (2).

In the two-stage algorithm, we change the first stage of mapping and partitioning. We still clone the nodes v with degree $d_v \geq T \times d_{avg}$ with percentage being less than or equal to a preset number ϵ. The constant factor T may vary from input to input according to degree distribution. But for these set of nodes, we do not add any auxiliary edge. We then partition the transformed graph and perform edge-fixing for an edge partition solution.

In the end, we look at these large degree nodes and place them into different clusters according to their incident edge placement. In the worst case, we need to place all $n\dot{\epsilon}$ nodes into all k cluster. Therefore, we add a constant in inequality (3).

In practice, we pick multiple pairs of (T, ϵ), perform mapping, partitioning and fixing for multiple times until we get satisfactory results. We try a range of values in ϵ from 0.05 to 0.25 and we pick the best partition result.

Large Edge/Node Ratio Graphs

Finally for data-affinity graphs that have large average degree, the edge/vertex load ratio ($edge/node = 2 * d_{avg}$) is high which implies the computation/memory ratio is high. These type of applications, for instance, the matrix multiplication application, if carefully scheduled to overlap computation and memory operations, can always achieve peak hardware computation peak. The matrix multiplication runs almost up to 80%-90% hardware computation peak in Kepler GPUs.

These type of applications are not sensitive to the total number of data loads since there is enough computation to hide latency. We use a third strategy for high average-degree data-affinity graphs. While partition the data-affinity graph, we ensure that in every computation cluster, edge/node ratio is above a threshold so that data overhead is hideble by careful instruction/thread scheduling. We describe the heuristic and other practical algorithm enhancement in Section 4.

Note for future references: the kdd balanced edge partition [Bourse et al. 2014], balanced graph partition [Krauthgamer et al. 2009]

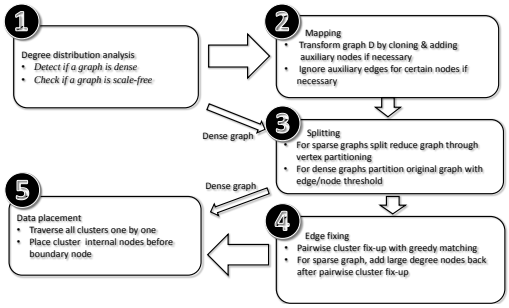

Figure 3. Overview of the practical partition framework – GRAPE

4. Practical Solution Framework

Section 3 shows the basic two-stage partitioning algorithms and provided analytical bound. Here we present the overall practical solution framework including the heuristic we used to enhance the partition results and ensuing data data layout transformation based on workload partitioning. We call it GRAPE – GRAph Partition on Edges framework.

We describe the whole framework in five major components as illustrated in Fig. 3: 1) degree distribution analysis, 2) mapping from one graph to another graph, 3) splitting graph into multiple parts, 4) edge-fixing after splitting on the transformed graph, and 5) data placement according to workload partitioning.

In the **degree distribution analysis** component, we analyze the degree distribution of the input data-affinity graph and determine graph type. If the graph is a dense graph, whose average degree is large $d_{avg} \geq d_{thresh}$, we perform edge partition under the constraint that every cluster's edge/node ratio $r_{e/v} \geq t$. We skip the mapping and the edge-fixing step and only go through splitting step. If the graph is relatively sparse $d_{avg} \leq d_{thresh}$, we perform mapping, splitting and edge-fixing steps. The **degree distribution analysis** component also guide the mapping component decision.

In the component of **mapping**, we map the original data-affinity D into another graph D', whose vertex number is twice as much as the edge number in D. There is direct correspondence between pairs of nodes in D' and edges in D. The mapping stage prepares for the splitting stage, in which D' can be split using graph vertex partition algorithms.

In the **splitting component**, based on the graph type result from *degree distribution analysis* component, we perform graph splitting either on transformed graph D' or original D. For splitting on transformed graph D', which is vertex partitioning, we use multi-level vertex partitioning. We use library METIS [Karypis and Kumar 1995] to perform vertex partitioning. The results of splitting component is further passed to the edge-fixing component.

For dense graphs, we perform edge partition directly. We iteratively select incident edges to place into a cluster based on the following priority criteria: 1) the edge needs to be an incident edge if there is any otherwise we pick an edge ran-

domly 2) the other end point of the incident edge has most connections to existing node in the cluster. At the initialization step, we pick the node that has largest degree and start picking edges to put into the same cluster using the priority criteria. We repeat this until one cluster is full and then we move to the next cluster. We find this approach works well for dense data-affinity graphs, for example, cliques. In fact, for a 2-way partition of a clique, it can be easily proved that this approach gives optimal communication cost.

In the **edge-fixing component**, depending on the cut set results of the partitioning on graph D', we fix the real edges that are contained in the cut set with the basic fixing approach discussed in Section 3.3. We still do edge fix-up for pair-wise clusters. But instead of arbitrarily choosing the destination, we give both clusters priority number for every real edge. Assume we have a real edge e and two clusters A, B, let $u \in A, v \in B$ correspond to two incident vertices of e, the priority number for cluster A of edge e, $priority_A(e) = |\{e_{v,w}, w \in A\}|$, that is the number of edges going from node v to any node in A. We define $priority_B(e)$ in the same way. We place the edge e into the cluster that has a higher priority number. Starting with the edge that has highest maximal priority number, we place edges into two clusters correspondingly until one cluster is full (with respect to the edges between these two clusters). Then we place the rest real edges into the other cluster.

Finally, in the **data placement component** we use the edge partitioning result to perform data placement to enhance data coalescing for data loads within the same cache-sharing thread cluster. We place data objects used in the same cluster near each other. We start by placing the data objects that are internal to a cluster together first. These nodes do not appear in more than 1 cluster. Next we place the boundary nodes of the cluster after internal nodes. We then move on to the next cluster that has largest number of incident nodes to the current one and perform data placement until we have determined the placement for all data objects. In initialization, we pick the first cluster as the one that has largest number of nodes.

Software Cache Data Pre-loading and Index Calculation In the case of using software cache, we need to explicitly manage the index (placement) of data objects in software cache. After workload partitioning with the data-affinity graph model, we get cluster of computation operations and the set of data objects associated with every cluster. By partitioning, we have made the number of data objects associated with every same number of computation operations as small as possible so that they can be placed in cache completely or almost at least completely for the set of operations that happen in parallel. When using software, we pre-load things into cache until software cache is full and perform computation. Then we pre-load the data objects for the next batch of operations. If necessary, we use *pathwidth* based algorithms [] which minimizes the number of active data objects at one time simultaneously.

Since the index of data object in cache is local to every cluster, we use smaller type for indices, *short* or *char* depending on the thread block sizes (*char* for thread block size < 256 and *short* for larger thread block sizes). The index calculation is done using *inspector-executor* [] approach at the inspector stage and the indices are applied in the executor stage.

Discussion We can choose over software cache or hardware cache in GPUs to use for the data a program reuses across threads. The advantage of software cache is that we can explicitly manage the placement and eviction of data objects in cache so that the data objects whose data reuse pattern do not fit the least recent use (LRU) policy can take advantage of fast on-chip memory as well. The advantage of hardware cache is that we do not need to worry about keeping track of indices of data objects in cache. We do not need to explicitly move data in and out of cache. For programs that have heavy data reuse, we use hardware cache since hardware cache can correctly cache the data that needs to be cached. For programs that have moderate data reuse and high concurrency (the number of threads active at the same time), hardware cache might not predict the eviction in the optimal way and/or can easily make cache polluted, we use software cache.

5. Evaluation

We evaluate our GRAPE framework with various important and practical kernels listed in Table 1. We conduct our experiments on a machine with NVIDIA Kepler GPU GTX680 with CUDA computing capability 3.0. It has 8 streaming multiprocessors with 192 cores on each of them. There are 65536 registers and 48KB shared memory on each SM. The host machine runs 64-bit Linux with kernel version 3.1.10 and CUDA 5.5.

For each benchmark, we perform code instrumentation to enable data and cache index pre-fetching if we use software cache. We perform mapping, splitting and edge-fixing with different parameters for different graphs and GRAPE returns workload partition and data placement order. To apply workload partitioning, we perform job swapping transformation [Zhang et al. 2011] or input data reordering (through changing format of the input file) since the input file contains a list of data objects and the program takes the order in the list as the order in memory. Examples include sparse matrix vector multiplication [] cusp and cfd from rodinia [Che et al. 2009]. By doing this, we can test and compare the actual effectiveness of graph partition without interference from job swapping (thread indirection) and etc. The detailed benchmark information is shown in Table 1.

b+tree B-tree is a tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. It is commonly used in databases

Table 1. Benchmark description.

Benchmark	Source	Graph	Max degree	Avg degree	Cache type	Added SLOC
b+tree	Rodinia	path	2	1.99	software	9
cfd	Rodinia	grid	4	3.91	software	144
gaussian	Rodinia	bipartite	16	16	software	8
particlefilter	Rodinia	clique	999	999	software	5
streamcluster	Rodinia	general	2	2	software	48
spmv	CUSP	bipartite	56181 39	20.57 (rail4284) 39 (qcd5_4)	hardware	64

and filesystems. A B+ tree can be viewed as a B-tree in which each node contains only keys. The data sharing is the back and forth movement over a path from root node to low level node. By caching the data on the path in software cache, We improved the performance of *b+tree*'s findK kernel by 1.67 times and fingRangeK kernel by 1.41 times.

cfd Computational fluid dynamics, usually abbreviated as CFD, is a branch of fluid mechanics that uses numerical methods and algorithms to solve and analyze problems that involve fluid flows. The data-affinity graph is a partially occupied grid. The data sharing occurs among nodes that are neighours on the grid. The maximal degree in CFD data-affinity graph is 4. The order of the particles in memory or thread groups is typically determined by Hilbert space filling curve [] since it is a good locality heuristic for dynamic scientific simulation code. We improved the performance of CFD upon the original partitioning obtained by Hilbert space filling curve by by over 18%.

gaussian Gaussian function (named after Carl Friedrich Gauss), often simply referred to as a Gaussian, is a function used to describe normal distribution, guassian filters and gaussian blurs etc. The data-affinity graph of Gaussian is very dense since it reuses data in cache significantly. However, the Gaussian application is also memory intensive, therefore we do not let data use hardware cache. With software cache (shared memory), the Gaussain performance is improved 2.19 times.

particlefilter Particle filters or Sequential Monte Carlo (SMC) methods are a set of on-line posterior density estimation algorithms that estimate the posterior density of the state-space by directly implementing the Bayesian recursion equations. SMC methods use a grid-based approach, and use a set of particles to represent the posterior density. This kernel is computation intensive, and its global memory accesses is very small. As a result, although we reduce global memory transactions by 99.6%, the performance improvement is only 4.8%.

streamcluster Data Stream clustering, given a sequence of n points in a metric space and an integer k, output k points in the sequence to minimize the sum of the distance of every point to its nearest neighborhood center. Data sharing occurs between different iterations that calculate distances of all points to a center point. Since all points are accessed for multiple times, we enhance cache sharing by modifying streamcluster to aggregate computation from every n iterations in 1 iteration, to explore sharing. Since aggregation has overhead, we set $n = 2$. We use software cache and streamcluster performance is improved by 19.9%.

spmv Sparse matrix-vector multiplication (SpMV) of the form $y = Ax$ is a widely used computational kernel existing in many scientific applications. The input matrix A is sparse. The data reuse happens when an element in the input vector x is accessed multiple times.

In fact, there is also reuse if we consider the data in the output vector y since we have do multiplication of a set of elements and then accumulate the result to an element in y. We build data-affinity graph using y and x elements since every A entry is used once and only once. To overcome the dependence problem, we use atomicAdd to accumulate the multiplication results to corresponding y element. It is shown that atomicAdd has been significantly improved for NVIDIA Kepler architecture. Massive atomic operations may not harm performance while it can improve performance [Egielski et al. 2014]. For fair comparison, we show the result of original non-atomic version, atomic version and our atomic version that used hardware cache. We evaluate GRAPE on two totally different types of matrices from *matrix market*: the matrix with scale-free graph and the matrix with non scale-free graph that has uniform degree distribution. For the former, we get 1.57 speedup and for the latter, we get 1.24 speedup. We show the details results in Table 2.

Table 2. Running time and load memory transactions of spmv: Org.T represents original memory transaction number, r represents scale-free matrix rail4284 and q represents matrix qcd5_4. Note that store transaction number is very small for spmv.

Input	Org	Atom	GRAPE	Org.T.	Atomic T.	GRAPE T.
r	2.8078	2.657	1.6831	3416680	3412525	1057988
q	0.508	0.3701	0.2966	672256	623616	180224

Software Cache V.S. Hardware Cache Software cache and hardware cache have their advantages and disadvantages. Hardware cache works well for data with heavy reuse and for data whose cache indices maintaining incurs large overhead. Software cache works for data that has moderate

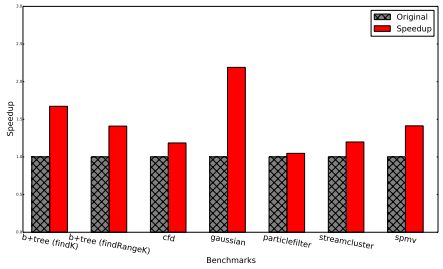

Figure 4. Overview of speedup for all graphs

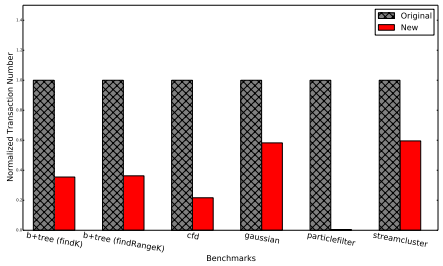

Figure 5. Overview of speedup for all graphs

or even small amount of data reuse, but it requires management of data placement and indexing in software cache. For all the seven kernels, six out of them us software cache while *spmv* uses hardware cache. That is because we used atomic writes for operations to the output vector y in $y = Ax$ and therefore sharing between x and y do not happen in the same memory location. Sharing over y happens within the buffer of atomic writes where spatial locality matters. If we use software cache for y, then we need to further aggregate the y value across thread blocks, which increases the complexity of transformation.

Small average degree V.S. large average degree For the benchmarks in Table 1, the *particlefilter* has largest average degree and thus has high edge/node ratio. Therefore, the program is computation intensive. We have seen little performance improvement for *particlefilter* since computation and memory is already overlapped very well, this confirms our conclusion about *particlefilter* above.

Overview We show the performance speedup for all seven kernels in Fig. 4. The baseline is the original program performance. The bar corresponding to *GRAPE* represents the performance of GRAPE's workload partitioning outcome. We show the normalized memory loads obtained using CUDA profiler in Fig. 5. The number of memory loads is normalized to the original number of memory loads.

As a summary, we find that partitioning workload for maximal cache performance is not as intimidating as the data-affinity graph looks like. Though the problem is NP hard itself, our proposed algorithm is approximated and work well for computation grid and scale-free data-affinity graphs. For most other benchmarks, the algorithm also yields good performance, which confirms the importance of using cache in GPU programs, despite the fact it is much smaller than CPU caches.

6. Related Work

Many compiler techniques are proposed to achieve better utilization of GPU memory. For affine loops, Baskaran *et al.* use a polyhedral compiler model to reduce non-coalesced memory accesses and bank conflicts in shared memory [Baskaran et al. 2008]. Jia *et al.* propose to characterize data locality and then guide GPU caching [Jia et al. 2012]. The limitation of these compiler methods is they cannot address dynamic data locality variation at runtime.

Some research uses hints provided by programmers to help compiler improve GPU memory performance. CUDA-lite tunes shared memory allocation via annotations [**?**]. hiCUDA seeks to automate shared memory allocation with the help of programmer specified directives [Han and Abdelrahman 2011].

There are also some studies to optimize some particular applications. Bell and Garland discuss data structures of sparse matrix-vector multiplication (spmv) for various sparse matrix formats [Bell and Garland 2008]. Choi *et al.* propose an automatic performance tuning framework for spmv [Choi et al. 2010]. Volkov and Demmel analyze the bottleneck in dense linear algebra and optimize its performance by improving on-chip memory utilization and etc. [Volkov and Demmel 2008]

The work closest to our study is software optimization for GPU memory performance. Zhang *et al.* propose to dynamically reorganize data and thread layout to minimize irregular memory accesses [Zhang et al. 2011]. Wu *et al.* also propose two data reorganization algorithms to reduce irregular accesses [Wu et al. 2013]. However, these papers do not address data sharing problem in GPU. Seo *et al.* use internal local memory in accelerators and GPGPUs to emulate hardware cache with software [Seo et al. 2009]. To achieve this goal, their method needs extra storage for cache tags and inserts instructions before each load/store instructions to check tags. Thus, it incurs significant overhead.

In the field of CPU memory optimization, the following studies are most relevant to this paper. Ding and Kennedy propose to use runtime transformation for improving memory performance of irregular programs [Ding and Kennedy 1999]. Bondhugula *et al.* introduce an automatic source-to-source transformation framework to optimize data locality, and they formulate data locality problem with polyhedral model [Bondhugula et al. 2008]. Udayakumaran *et al.* propose a software based scratch-pad memory management scheme for embedded processors [Udayakumaran et al. 2006].

7. Conclusion

Since hundreds of thousands of threads can execute simultaneously in the GPU, it is very difficult to exploit shared cache benefits even the program locality is good. The non deterministic feature of thread execution in GPU makes the situation even worse.

To address this problem, we formulate a generic workload partitioning model that systematically exploits the complexity and approximation bound for optimal cache sharing among GPU threads in this paper. Our experiments show that our method can improve data sharing and thus performance significantly for GPU benchmarks. Our exploration in this paper demonstrates that it is possible to utilize GPU cache efficiently without significant programming overhead or ad-hoc application-specific implementation.

References

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 the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 225–234, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3. doi: 10.1145/1375527.1375562. URL http://doi.acm.org/10.1145/1375527.1375562.

N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375595. URL http://doi.acm.org/10.1145/1375581.1375595.

F. Bourse, M. Lelarge, and M. Vojnovic. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1456–1465, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2956-9. doi: 10.1145/2623330.2623660. URL http://doi.acm.org/10.1145/2623330.2623660.

S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-5156-2. doi: 10.1109/IISWC.2009.5306797. URL http://dx.doi.org/10.1109/IISWC.2009.5306797.

J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 115–126, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-877-3. doi: 10.1145/1693453.1693471. URL http://doi.acm.org/10.1145/1693453.1693471.

C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 229–241, New York, NY, USA, 1999. ACM. ISBN 1-58113-094-5. doi: 10.1145/301618.301670. URL http://doi.acm.org/10.1145/301618.301670.

I. J. Egielski, J. Huang, and E. Z. Zhang. Massive atomics for massive parallelism on gpus. In *Proceedings of the 2014 International Symposium on Memory Management*, ISMM '14, pages 93–103, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2921-7. doi: 10.1145/2602988.2602993. URL http://doi.acm.org/10.1145/2602988.2602993.

T. Han and T. Abdelrahman. hicuda: High-level gpgpu programming. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):78–90, Jan 2011. ISSN 1045-9219. doi: 10.1109/TPDS.2010.62.

W. Jia, K. A. Shaw, and M. Martonosi. Characterizing and improving the use of demand-fetched caches in gpus. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 15–24, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1316-2. doi: 10.1145/2304576.2304582. URL http://doi.acm.org/10.1145/2304576.2304582.

G. Karypis and V. Kumar. Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

R. Krauthgamer, J. S. Naor, and R. Schwartz. Partitioning graphs into balanced components. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 942–949, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics. URL http://dl.acm.org/citation.cfm?id=1496770.1496872.

J. Liu, W. Ding, O. Jang, and M. Kandemir. Data layout optimization for gpgpu architectures. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 283–284, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. doi: 10.1145/2442516.2442546. URL http://doi.acm.org/10.1145/2442516.2442546.

S. Seo, J. Lee, and Z. Sura. Design and implementation of software-managed caches for multicores with local memory. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 55–66, Feb 2009. doi: 10.1109/HPCA.2009.4798237.

M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 91–102, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. doi: 10.1145/781131.781142. URL http://doi.acm.org/10.1145/781131.781142.

S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, May 2006. ISSN 1539-9087. doi: 10.1145/1151074.1151085. URL http://doi.acm.org/10.1145/1151074.1151085.

V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9. URL http://dl.acm.org/citation.cfm?id=1413370.1413402.

B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 57–68, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. doi: 10.1145/2442516.2442523. URL http://doi.acm.org/10.1145/2442516.2442523.

Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-

3. doi: 10.1145/1806596.1806606. URL http://doi.acm.org/10.1145/1806596.1806606.

E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 369–380, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1. doi: 10.1145/1950365.1950408. URL http://doi.acm.org/10.1145/1950365.1950408.