Locality-Aware Software Throttling for Sparse Matrix Operation on GPUs

Yanhao Chen* *Rutgers University*
Ari B. Hayes* *Rutgers University*
Chi Zhang *University of Pittsburgh*
Timothy Salmon *Rutgers University*
Eddy Z. Zhang *Rutgers University*

Abstract

This paper tackles the cache thrashing problem caused by the non-deterministic scheduling feature of bulk synchronous parallel (BSP) execution in GPUs. In the BSP model, threads can be executed and interleaved in any order before reaching a barrier synchronization point, which requires the entire working set to be in cache for maximum data reuse over time. However, it is not always possible to fit all the data in cache at once. Thus, we propose a locality-aware software throttling framework that throttles the number of active execution tasks, prevents cache thrashing, and enhances data reuse over time. Our locality-aware software throttling framework focuses on an important class of applications that operate on sparse matrices (graphs). These applications come from the domains of linear algebra, graph processing, machine learning and scientific simulation. Evaluated on over 200 real sparse matrices and graphs that suffer from cache thrashing in the Florida sparse matrix collection, our technique achieves an average of 2.01X speedup, a maximum of 6.45X speedup, and a maximum performance loss $\leq$5%.

1 Introduction

Operations on sparse matrix and graph are important for solving linear algebra and optimization problems that arise in data science, machine learning, and physics-based simulation. In this paper we focus on a fundamental sparse matrix operation that relates an input vector x with an output vector y. Let $\mathbf{x}$ be an $n \times 1$ vector, $\mathbf{y}$ be an $m \times 1$ vector, and A be a $m \times n$ matrix, the relation between y and x is defined as $\mathbf{y} = A\mathbf{x}$, where

$$y_i = reduce\_op\{A_{ik} \bigodot x_k\},\ 1 \leq k \leq n.$$

When the operator *reduce_op* is *sum* and the binary operator $\bigodot$ is multiplication, the operation is sparse matrix vector multiplication (SpMV). When the operator *reduce_op* is *min* and the binary operator $\bigodot$ is $+$, the operation is an iterative step in the single source shortest path (SSSP) problem [13]. An example is shown in Figure 1.

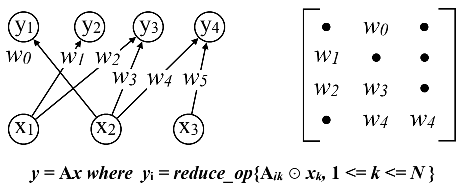


Figure 1: A Fundamental Sparse Matrix Operation

However, poor data reuse is often a problem when running sparse applications on GPUs. Throttling is a useful technique to improve data reuse. Unlike other locality enhancement techniques that focus on spatial data reuse on many-core, for instance, the memory coalescing techniques [33], throttling improves data reuse over time by limiting the number of actively executed tasks.

Throttling prioritizes the execution of the tasks that reuse the data in the cache over those that do not reuse the data in the cache. Figure 2 shows an example of how throttling improves cache data reuse. Assuming the cache capacity is 4, in the original case, the cache cannot hold all the data elements in the execution list which will inevitably cause cache (capacity) misses. Throttling helps by dividing the execution into two phases and scheduling one phase after another. Data elements in each phase can now fit into cache and be fully reused so that no cache (capacity) misses will occur.

Throttling for GPU has been studied extensively in the hardware context. Rogers and others [27] discovered that limiting the number of active wavefronts (warps) enhances cache data reuse over time and alleviates cache thrashing. The DYNCTA framework [19] limits the number of CTAs for memory intensive applications and results in better cache performance. The work by Chen and others [8] augmented cache bypassing with a dy-

*First authors Chen and Hayes have made equal contributions to this work and are listed alphabetically.

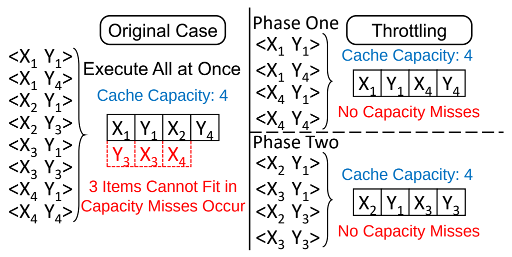


Figure 2: Throttling Example

namic warp-throttling technique to improve both cache performance and energy efficiency. However, all these prior throttling techniques on GPUs have been developed as hardware modifications.

In this paper, we present a software throttling framework that targets irregular applications operating on sparse data. Our software throttling framework will first divide the entire workload into multiple partitions such that the working set of each partition fits into the cache and the data communication between different partitions is minimum (we will refer to each partition as *cache-fit partition* or *cache-fit work group* throughout this paper). Then we schedule the cache-fit partitions and let each of them be processed independently to ensure throttling.

There are three main challenges for realizing software throttling. First, the traditional work partition models focus on minimizing data reuse among different partitions with load-balancing constraints [2, 6, 29]. However, cache-fit work partitioning is not necessarily load-balanced, it should be data-balanced across different partitions. Second, inappropriate scheduling of cache-fit partitions might result in low execution pipeline utilization. For each of the cache-fit partitions that have low data reuse, there may not be enough tasks running concurrently, which will make the execution pipeline units not fully utilized and degrade the computation throughput. Last, reducing the overhead of software throttling is important and yet challenging, especially for finding minimum communication *cache-fit* partitions, which is the most time-consuming step in software throttling.

To tackle these challenges, we propose the three following techniques. To obtain *cache-fit* partitions, we develop an efficient **data-balanced** work partition model. Our partition model can balance data while minimizing the communication cost among different partitions. We also introduce a **split-join** scheduling model to take advantage of the trade-off between throttling and throughput. The *split-join* scheduling model adaptively merges partitions to avoid low execution pipeline utilization and/or use a concurrent queue based implementation for relaxed barrier synchronization. We reduce the partition overhead by a coarse-grained partition model which was built upon a multi-level partition paradigm. Instead of partitioning the original work matrix (graph), our model partitions a coarsened matrix (graph) which can significantly reduce the partition overhead while maintaining similarly good partition quality.

Our throttling technique is a pure software based implementation. It is readily deployable and highly efficient. Evaluated over 228 sparse matrices and graphs from Florida matrix collection [11] - the set of matrices which suffer from cache thrashing (their working set cannot entirely fit into the L2 cache on the Maxwell GPU and Pascal GPU we tested), our software throttling method can achieve an average 2.01X speedup (maximal 6.45X speedup).

As far as we know, this is the first work that systematically investigates **software throttling** techniques for GPUs and is extensively evaluated on real sparse matrices and graphs. The contribution and the outline of our paper is summarized as follows:

- We introduce an analytical model named *data-balanced* work partition for locality-aware software throttling. Efficient heuristics are developed to achieve (near-)minimum communication *cache-fit* work partitions that can be further scheduled to alleviate GPU cache thrashing (Section 2).

- We exploit the trade-off between cache locality and execution pipeline utilization and provide a set of practical *cache-fit* work group scheduling policies based on *adaptive merging* and *concurrent dequeuing*. We discuss the advantages/disadvantages, the applicability, and the effectiveness of each scheduling policy in different settings. (Section 3).

- Our method requires *no hardware modification*. It is low overhead and readily deployable. We introduce efficient overhead control mechanisms for graph(matrix)-based work partition. (Section 4).

- We conduct a *comprehensive data analysis* for over 200 large real sparse matrices(graphs). Our framework in particular works well for the set of sparse matrices that have large working sets and suffer from high GPU cache contention (Section 5).

2 Data-Balanced Work Partition

Our software throttling framework first divides the entire workload into cache-fit partitions. A cache-fit partition's working set fits into the cache such that it will not cause any cache capacity miss. This section presents the concept and methodology of data-balanced work partition.

2.1 Graph Representation

In this paper, we focus on a fundamental operation in sparse linear algebra and optimization applications. It is defined as follows. Assume we have an $m \times n$ matrix A, an $n \times 1$ vector x, and an $m \times 1$ vector y such that:

$$y_i = reduce\_op\{A_{ik} \bigodot x_k\}, 1 \leq k \leq n \quad (1)$$

The operator $\odot$ is a binary operator, and the operator *reduce_op* is a reduction operator. When $\odot$ is product $\times$ and *reduce_op* is *sum*, the operation is a sparse matrix vector multiplication (SpMV). When $\odot$ is plus $+$ and *reduce_op* is *min*, the operation is a *min/product* step in the single source shortest path (SSSP) problem.

We represent a computation unit as a 2-tuple (x_j, y_i) which represents (1) one binary $\odot$ operation between x_j and A_{ij} , and (2) one step in the reduction operation *reduce_op* for obtaining y_i. We only focus on vector x and y, since the matrix entries will be used only once in Equation (1).

We represent the entire workload as a *2-tuple* list. Using a graph representation, each data element in the *2-tuple* is modeled as a vertex and each tuple is modeled as an edge that connects the corresponding two vertices. Performing a work partition is essentially performing an edge partition on the graph, as illustrated in Figure 3.

2.2 Data-Balanced v.s. Load-Balanced

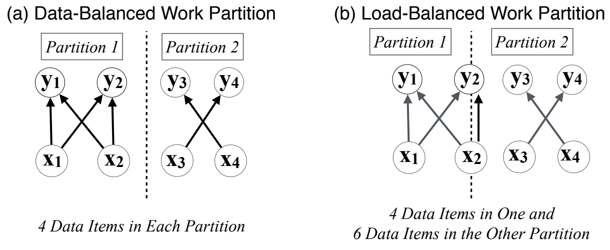
We formally define the **data-balanced** *work partition* model. The input is a list of *2-tuple* modeled as a work graph and the output is a set of minimum-interaction work partitions such that the number of unique vertices, which represent data elements, in every work partition is less than or equal to the cache capacity.

In contrast to prior load-balanced work partition, we perform data-balanced work partition. We denote this problem as a *Vertex-balanced Edge-Partition (V-EP)* model and we give the definition below:

Definition 2.1.
Vertex-balanced Edge-Partition (V-EP) Problem *Given a graph $G = (V,E)$ with the set of vertices V and the set of edges E, and vertex capacity constraint T. Let $x = \{e_1, e_2, ...e_k\}$ denote a partition of the edges of G into k disjoint subsets, and let $V(e_i)$ denote the set of unique vertices in e_i. $\forall n \in V$, let $P(n)$ denote the number of subsets that n's incident edges fall into. We optimize the total vertex replication cost:*

$$\begin{aligned} &\underset{x}{\text{minimize}} \quad R(x) = \sum_{n \in V} (P(n) - 1) \\ &\text{subject to} \quad \forall i \in [1..k], |V(e_i)| \leq T \end{aligned} \quad (2)$$

In prior work, the *Edge-balanced Edge-Partition* (E-EP) problem has been well studied particularly in the distributed graph processing setting [6, 14] and also for balancing workloads in GPU [25, 26]. However, the V-EP problem is not. Both the V-EP and E-EP problems minimize vertex replication cost, while the E-EP model aims to balance the load among processors in space, and the V-EP model aims to alleviate cache thrashing and maximize data reuse over time.



(a) Data-Balanced Work Partition
Partition 1
Partition 2
y1
y2
y3
y4
x1
x2
x3
x4
4 Data Items in Each Partition
(b) Load-Balanced Work Partition
Partition 1
Partition 2
y1
y2
y3
y4
x1
x2
x3
x4
4 Data Items in One and
6 Data Items in the Other Partition

Figure 3: Data-Balanced v.s. Load-Balanced

We use an example in Figure 3 to illustrate the difference between the V-EP work partition and the E-EP work partition. Assuming the cache capacity is 4, Figure 3 (a) shows a 2-way V-EP work partition: one partition has 4 edges and the other has 2, the unique vertices of both (4 vertices) fit into cache. Figure 3 (b) shows another 2-way E-EP partition: Each partition has 3 edges, however *partition 2* has 6 unique vertices and do not fit into cache. Thus the E-EP model might exacerbate rather than alleviate the cache thrashing problem.

2.3 Partition Framework

We propose a data-balanced work partition framework that ensure the working set of each partition is of the same size and in the meantime the data reuse across different partitions is reduced as much as possible.

Our partition framework is a recursive *bisection* framework. Bisection is a *2-way* balanced edge partition that ensures minimum vertex replica between two equal-size edge partitions. The optimal *bisection* is a well studied problem [25]. We take the advantage of the *bisection* method and perform hierarchical partitioning.

During the recursive partition process, the framework bisects a sub-graph that has more unique vertices than specified by the capacity constraint. It keeps bisecting until no such sub-graph exists.

We use a tree data structure to keep track of the obtained sub-graphs. Starting from the root node that represents the entire work graph, the framework bisects the corresponding graph and generates two children nodes: each of the child nodes corresponds to a sub-graph that contains half of the edges from the parent node. If either or both children nodes violate the capacity constraint, either or both will be added to the list of sub-graphs that need to be further bisected. The process repeats until all leaf nodes become cache-fit work partitions.

The detailed algorithm is listed below in Algorithm 1. The *data-balanced work partition* (DBWP) procedure takes the work graph G and the cache capacity constraint T as input, and generates a set of cache-fit partitions P as output. The *bisect* function in Algorithm 1 we adopted is based on the best existing balanced edge partition algorithm named SPAC [26, 24] by Li and others. We will discuss the implementation details and the overhead control mechanisms of the *bisect* function in Section 4.

Algorithm 1 Data-Balanced Work-Partition (DBWP)

```
Input: work graph $G$, cache capacity $T$
Output: cache-fit partition set $P$
1: procedure DBWP($G$, $T$, $P$)
2:     if $|G.data\_elements| > T$ then
3:         ($lchild$, $rchild$) = bisect($G$)
4:         DBWP($lchild$, $T$, $P$)
5:         DBWP($rchild$, $T$, $P$)
6:     else
7:         add $G$ to $P$
8:     end if
9: end procedure
```

We use an example to illustrate the *DBWP* procedure in Figure 4. In this example, the graph has 8 edges and 8 vertices, and the cache capacity constraint is 4. Performing *bisect* for the sub-graph represented by the tree root node, we obtain two sub-graphs each of which has 4 edges. The vertex replica cost is optimum: 2, since two nodes y_2 and x_2 appear in both partitions.

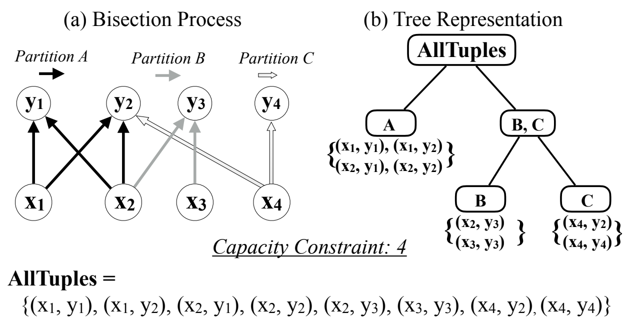


Figure 4: Hierarchical Bisection Example

The first sub-graph A in Figure 4 (a) has 4 unique vertices and does not violate the capacity constraint, we do not perform further bisection on sub-graph A. The other sub-graph, however, has 6 unique vertices and does not fit into the cache. Therefore we perform the second bisection and obtain partitions B and C where the vertex replica cost is optimum (0 in this case). At this point, there is no sub-graph that does not fit into the cache, therefore we terminate the bisection process. The tree representation is shown in Figure 4 (b).

3 Cache-Fit Partitions Scheduling

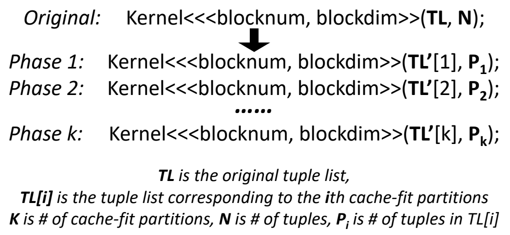
DBWP model outputs a set of *cache-fit* partitions. All these partitions need to be processed independently to minimize cache-thrashing interference. However, naive scheduling of these partitions might result in low execution pipeline utilization. In this section, we introduce four different *Cache-Fit Partitions* Scheduling methods: *Cache-Fit* Scheduling (**CF**), *Cache-Fit Queue* Scheduling (**CF-Q**), *Split-Join* Scheduling (**SJ**) and *Split-Join Queue* Scheduling (**SJ-Q**).

CF works well when all cache-fit partitions have high data reuse. **SJ** is good for cases when the sparsity structure is already known, for instance, pruned deep leaning neural networks. Both **CF-Q** and **SJ-Q** can loosely enforce throttling and provide a better performance.

3.1 Cache-Fit Scheduling

A straightforward way to isolate the computation of different cache-fit partitions is to assign each partition a single *kernel* function and execute these kernels one by one. A *kernel* is a function that is executed on GPU. All threads within a GPU kernel will need to finish before the entire kernel complete – there is a strict barrier between different GPU kernels. Moreover, between two consecutive kernels, the data in the cache will be invalidated.

Here, we propose **CF** which separates the original kernel functions into multiple kernels, while the number of which is determined by the number of cache-fit partitions given by **DBWP** model. The code of the kernel function for each cache-fit partition is the same. The only difference is the input to each kernel. **CF** ensures that the data in the cache is fully reused before it was evicted from the cache within each cache-fit partition.


Original: Kernel<<<blocknum, blockdim>>>(**TL**, **N**);

Phase 1: Kernel<<<blocknum, blockdim>>>(**TL'**[1], $\mathbf{P_1}$);
Phase 2: Kernel<<<blocknum, blockdim>>>(**TL'**[2], $\mathbf{P_2}$);
……
Phase k: Kernel<<<blocknum, blockdim>>>(**TL'**[k], $\mathbf{P_k}$);

TL is the original tuple list,
***TL[i]** is the tuple list corresponding to the **i**th cache-fit partitions*
***K** is # of cache-fit partitions, **N** is # of tuples, $\mathbf{P_i}$ is # of tuples in TL[i]*

Figure 5: Kernel Splitting for Cache-Fit Scheduling

We show the idea of **CF** in Figure 5. The input *2-tuple* task list TL is split into k *2-tuple* lists TL', which corresponds to each of the cache-fit partitions. Each new tuple list will be processed by a single kernel.

3.2 Cache-Fit Queue Scheduling

CF makes sure that each cache-fit partition will be processed by one kernel. Although **CF** can provide good throttling performance for a lot of matrices, this scheduling method may cause low execution pipeline utilization for the type of matrices whose data reuse is low. For example, for a given cache size T and average data reuse ratio r for a cache-fit partition, the total work is $T * r/2$ using our work graph model. The variable T is fixed for a given architecture. If r is low, the number of concurrent tasks in one cache-fit partition p is low and may not keep the execution pipeline busy.

To avoid this problem, we propose **CF-Q**, which processes the whole tuple list in a single kernel instead of one invocation per cache-fit partition. However, using a single kernel means that elements in one cache-fit partition have no guarantee to be executed without any interference. To enable throttling, we set up a FIFO queue before launching the kernel. Each queue entry corresponds to a chunk of tuples so that adjacent chunks are from the same cache-fit partition. A warp automatically fetches a chunk from the queue and process the tuples from that chunk. We show an example of how **CF-Q** works in Figure 6.

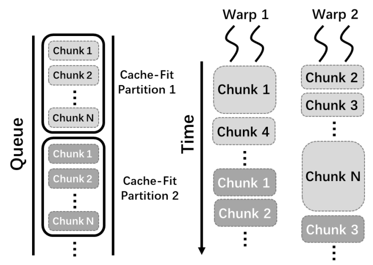


Figure 6: Queue Based Scheduling Example

Unlike **CF** which has explicit barriers to strictly enforce the independent execution of different cache-fit partitions, **CF-Q** uses no barrier. It is possible that the last chunk in one partition and the first chunk in the next partition are fetched within a very short time period. In Figure 6, chunk 1 and chunk 2 from partition 2 will be running concurrently with chunk N from partition 1. However, **CF-Q** can still provide relaxed barriers between different partitions since chunks from the same cache-fit partition in the queue will always be retrieved in consecutive time periods so that no following partitions can be executed before previous one starts. The pseudo code of **CF-Q** is provided in Algorithm 2.

3.3 Split-Join Scheduling

Split-Join (**SJ**) is another method that exploits the tradeoff between locality and execution pipeline utilization. **SJ** dynamically merges the *cache-fit* partitions that has low data reuse or combines low data reuse partitions with a high data reuse partition that is less likely to be interfered. **SJ** first constructs the tree structure that represents the hierarchical cache-fit partitions which we discussed

Algorithm 2 Cache-Fit Queue Scheduling

```
Input: cache-fit partition set P
 1: procedure CF-Q(P)
 2:     for each partition p in P do
 3:         insert p into queue Q
 4:     end for
 5:     Kernel(Q)
 6: end procedure
 7: procedure KERNEL(Q)
 8:     while Q is not empty do
 9:         I ← next queue item (chunk) from Q
10:         process I[laneID]
11:     end while
12: end procedure
```

in Section 2.3, we will refer to this as *SJ-tree*. **SJ** merges sibling nodes in the *SJ-tree* conditionally in a bottom-up manner. We only consider recombining sibling nodes as the sibling nodes have better data sharing than non-sibling nodes and the merging is logarithmic time.

SJ is performed with a fast online profiling process. We define a *profiling pass* as a profiling of all the nodes at one level of the *SJ-tree* which comprises one traversal of the entire work graph. So, the entire profiling process will take d profiling passes, where d is the depth of the *SJ-tree*. It takes at least $log(k)$ and up to k profiling passes for any given *SJ-tree*, where k is the number of leaf nodes in the tree. The lower bound $log(k)$ is reached when the binary tree is balanced and has $log(k)$ levels. Moreover, in the worst case scenario, when the tree is not balanced and at every level there is at most one leaf node, k profiling passes are needed. We run every work partition that corresponds to a leaf node in the *SJ-tree* in *stand-alone* mode and record the running time.

We use the first d iterations of the linear algebra and optimization applications to collect information for profiling. Since those applications we tested take between 50 and 22,000 iterations to converge, the overhead of profiling can be amortized. For example, *G3_circuit* need 5 iterations for profiling, and the total profiling time for Conjugate Gradient (CG) solver takes 0.017 s. The running time for CG with 22824 iterations is 94.331 s which gives us 0.018% profiling overhead. In practice, among all the matrices we used in the experiment, we found that the SJ-tree had at most 8 levels and thus required at most 8 passes for profiling.

The tree node merging problem can be defined as a tree-based weighted set cover problem. Merging two sibling nodes is as if choosing their parent node. The problem becomes how to find a subset of tree nodes $\boldsymbol{P}$ that will cover all possible cache-fit partitions (leaf nodes) while minimizing the overall running time:

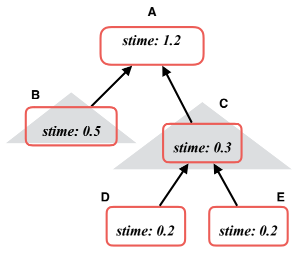
$$\begin{aligned} \text{minimize} \quad & \sum_{x \in P} c(x) \\ \text{subject to} \quad & \bigcup_{x \in P} S(x) = L \end{aligned} \tag{3}$$

where L is the set of all leaf nodes, P is the subset of the tree nodes (both leaf and non-leaf nodes) we want to find, $c(x)$ is a cost function that denotes the standalone running time of node x and S(x) is a set function that returns all leaf nodes of the subtree under node x.

Algorithm 3 Tree Recombination

```
Input: SJ-tree root
Output: optimal running time of SJ-tree root
 1: procedure TREERECOMB(root)
 2:     return BESTCONFIG(root)
 3: end procedure
 4: procedure BESTCONFIG(r)
 5:     left_t = BESTCONFIG(r.leftChild)
 6:     right_t = BESTCONFIG(r.rightChild)
 7:     this_node_t = r.stime
 8:     r.btime = min(left_t + right_t, this_node_t)
 9:     return r.btime
10: end procedure
```

We develop a linear time algorithm that is capable of finding the optimum solution for the *tree-based set cover* problem. The algorithm processes the tree in post-topological order. Every node is associated with an attribute *btime* and an attribute *stime*. A sub tree's optimum time *btime* (annotated as an attribute of its root node) is the minimum of the two items: its root node's standalone running time *stime* and the summation of its two children subtree's *btime*. For a leaf node, its *btime* is the same as its standalone running time *stime*. The pseudo code is provided in Algorithm 3 together with an example in Figure 7. This process identifies the best set cover of the SJ-tree and determines how to recombine cache-fit partitions into every GPU kernel. **SJ** can achieve high execution pipeline utilization without sacrificing cache benefits (as data reuse is low for these low pipeline utilization cases).



A.btime	*min*(A.stime, B.btime + C.btime)	0.8
B.btime	B.stime	0.5
C.btime	*min*(C.stime, D.btime + E.btime)	0.3
D.btime	D.stime	0.2
E.btime	E.stime	0.2

{B, C} is chosen as the best configuration.

Figure 7: Tree Node Recombination Example

3.4 Split-Join Queue Scheduling

SJ dynamically merges cache-fit partitions that has low data reuse to ensure high execution pipeline utilization and good throttling performance. However, although **SJ** can provide Strict Barriers between different (merged) partitions, **SJ** cannot guarantee the execution order of those cache-fit partitions inside the merged partitions.

We propose **SJ-Q** which uses the idea of **CF-Q** that places cache-fit partitions in one merged work group (kernel) into a queue and each kernel will be using one independent queue. **SJ-Q** can provide both strict barriers between different merged partitions and also relaxed barriers between cache-fit partitions from the same merged partition. In the mean time, it inherits the advantage of **SJ** that avoids low execution pipeline utilization.

Sched.	Pipeline Util.	Prof.	Barrier	Queue	Code Change
CF	Low	No	Strict (S)	No	No
CF-Q	High	No	Relaxed (R)	Yes	Yes
SJ	High	Yes	Strict	No	No
SJ-Q	High	Yes	S/R	Yes	Yes

Table 1: Comparison of Four Scheduling Methods: Sched. refers to Scheduling Method, Prof. refers to if profiling is needed, and Util. refers to utilization.

Summary CF enforces strict barriers between different cache-fit partitions to ensure throttling. However, low execution pipeline utilization may happen which can degrade the computation performance; **CF-Q** uses a queue based method that can fully utilize the execution pipeline and loosely enforce barriers between consecutive cache-fit partitions; **SJ** merges those low data reuse partitions into one based on a tree set cover algorithm and online profiling; **SJ-Q** enforces strict barriers between different merged partitions and relaxed barriers between different cache-fit partitions within one merged partition.

We summarize the features of four scheduling methods in Table 1. All methods ensure good throttling performance while different methods impose different levels of barrier synchronization and code change overhead.

4 Implementation

We perform the adaptive overhead control mechanisms and data reorganization to make our software throttling method more efficient. We reduce the overhead of bisection in the **DBWP** Model, which is the most time-consuming part. In particular, we focus on two bisection algorithms: 1) Coarsened Bisection built upon the SPAC model by Li and others [25], and 2) K-D tiling built upon the k-d tree geometric space partitioning method [5].

We also use CPU-GPU pipelining to make the scheduling overhead transparent [33]: the CPU determines the best schedule while GPU is doing the actual computation. We improve the kernel performance by transforming data layout after we get cache-fit partitions such that the data access within the same kernel is coalesced as much as possible.

4.1 Adaptive Overhead Control

Coarsened Bisection *Coarsened Bisection* is based on SPAC [25] an effective sequential edge partition model. SPAC relies on a multi-level partition paradigm, in which, a graph is coarsened, partitioned, and refined/uncoarsened level by level. In *Coarsened Bisection*, we reduce the overhead of SPAC by eliminating the last few levels of refinement steps. We discovered that the last few coarsened levels (five to seven levels) of refinement stages in the multilevel partitioning scheme, if omitted, do not lead to much performance difference for our software throttling methods. While at the same time, a large amount of partition over can be saved.

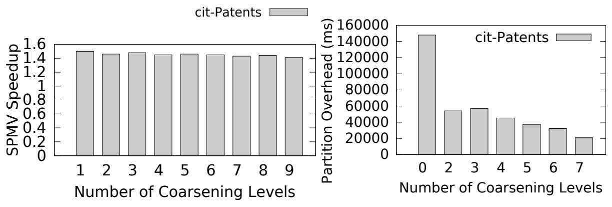


Figure 8: Trade-off between Eliminated Refinement Levels and Scheduling Overhead/Benefits

We show the trade-off between the number of levels where the refinement step is eliminated, and the SPAC partition overhead, and the *SpMV* speedup when applying the SPAC for scheduling, for the sparse matrix cit-patents [11] in Figure 8. It can be seen from the figure that by eliminating the last five to seven levels of refinement, the overhead is reduced by up to 7.5x, while the *SpMV* speedup only changed from 1.5x to 1.4x.

The detailed algorithm is showed in Algorithm 4. Notice that the input graph is already an coarsened graph, since we can perform first level coarsening while reading data from file. We also parallelize the merging phase in the coarsening phase to further reduce overhead. The merging phase is mainly for reconstructing the coarsened graph in each level and is amenable to parallelization.

Algorithm 4 Coarsened Bisection

```
Input: Coarsened Graph $G$
Output: Partition $P$
 1: procedure COARSENBISECT($G$)
 2:     // we call a set of edges - an entity
 3:     build entity based adjacent list $L$ of $G$
 4:     for $level \in \{1, \ldots, maxLevel\}$ do
 5:         sort $L$ by entity degree
 6:         for each entity $e$ do
 7:             merge $e$ with its heaviest avaliable neighbor $ne$
 8:         end for
 9:         build coarsened $L$ by results from above step
10:     end for
11:     build coarsened graph $G'$ from $L$
12:     $P \leftarrow$graphPartition($G'$)
13: end procedure
```

K-D Tiling Another bisection method we adopted is a tiling based method: *K-D Tiling*. Since any graph can be converted into a sparse matrix representation, we treat the partition as a partition in a geometric two-dimensional space. This method is similar to the k-d tree structure [5] used for partitioning a k-dimensional space. Every non-leaf node in a k-d tree represents a division point along a single spatial dimension, recursively splitting the space's points into two groups.

This partitioning method has even lower overhead than *Coarsened Bisection*. Each split can be performed in O(n) average time via the *quickselect* algorithm [16], and the number of rounds of splitting is logarithmic. However, unlike *Coarsened Bisection*, the tiling approach does not consider connectivity of the graph, and so it generates inferior results. This trade-off makes *Coarsened Bisection* preferable in applications where its overhead can be hidden via amortization, for instance, in optimization problems, and the K-D tiling method is better for overhead-sensitive applications.

4.2 Data Reorganization

After we perform *Data-Balanced Work-Partition* on the work graph, we reorganize the data in memory according to cache-fit partitions for efficient memory coalescing. We prioritize the partition that has the smallest amount of unique data – indicating a high data reuse if the amount of work in each partition is the same. We iterate over each partition's tuple list, and place all their non-boundary vertices (vertices that only appear in one kernel) consecutively in memory using *data packing* [12]. After non-boundary vertices for each partition have been processed, we process boundary vertices. The data reordering algorithm is briefly described in Algorithm 5.

5 Evaluation

We perform experiments on two platforms: an NVIDIA *GTX 745* GPU with Intel Core i7-4790 CPU and an NVIDIA *TITAN X* GPU with Intel Xeon CPU E5-2620. The GPU configurations are detailed in Table 2. We evaluate our techniques using important real-world workloads including sparse linear algebra, neural networks, and graph analytics.

Table 2: Experimental Environment

GPU Model	Titan X	GTX 745
Architecture	Pascal	Maxwell
Core #	5376	576
L2 Cache	3MB	2MB
CUDA version	CUDA 8.0	CUDA 8.0

Algorithm 5 Data Remapping

```
Input: Original Partition Set $P$, Boundary Vertex Set $B$
Output: Reordered Data $D$
 1: procedure DATAREMAPPING($P, D$)
 2:     for each vertex $v$ in partition $p$ of $P$ do
 3:         if $v \notin B$ then
 4:             $unique[p]$++
 5:         end if
 6:     end for
 7:     P' = rank(P, $unique$[P]); // Rank $P$ by $unique$[]
 8:     // Assign non-boundary nodes
 9:     for each vertex $v$ in partition $p$ of $P'$ do
10:         if !$boundary[v]$ and $v$ is not in $D$ then
11:             append $v$ to $D$
12:         end if
13:     end for
14:     // Assign boundary nodes
15:     for each vertex $v$ in partition $p$ of $P'$ do
16:         if $boundary[v]$ and $v$ is not in $D$ then
17:             append $v$ to $D$
18:         end if
19:     end for
20: end procedure
```

Sparse Linear Algebra Workloads We use sparse matrix vector multiplication (SpMV) and the conjugate gradient solver (CG). We present performance and sensitivity analysis, as well as the effectiveness of overhead control.

Neural Networks We use a pruned form of AlexNet [15]. The pruned neural network is essentially sparse matrix operation.

Graph Processing Workloads We use two graph processing benchmarks: the Bellman-Ford (BMF) and PageRank (PR) programs [21]. Bellman-Ford takes a weighted graph as input and iteratively calculates every node's distance – an important, basic operation used in path and network analysis applications. PageRank takes a weighted graph as input and calculates the importance of every node based on its incoming links.

Computational Fluid Dynamics Workloads We use the CFD benchmark from the Rodinia benchmark suite [7]. The CFD solver is an unstructured grid finite volume solver for the three-dimensional Euler equations for compressible flow. The CFD benchmark is already highly optimized in terms of data layout [9]. We use three mesh input sets from Rodinia [7].

5.1 Sparse Linear Algebra

SpMV We treat the sparse matrix as a bipartite graph, as described in Section 2.1, and then apply our techniques . We use the SpMV kernel function from the cusp library[10] and the matrix format is COO.

Of the 2757 matrices in the University of Florida collection [11], we extract those where the working set cannot fit entirely into the L2 (last-level) cache, which leaves us with 228 matrices on GTX 745, and 192 on Titan X. Though we optimize for the L2 cache, our techniques can be generalized to other caches.

The performance summary for *SpMV* across these matrices is shown in Figure 9. We also include Org+R, which applies the data reorganization scheme described in Section 4.2 to the original program to optimize memory coalescing. Memory coalescing is a technique for enhancing spatial locality [33], which is orthogonal to our technique proposed in this paper. As our work also performs memory coalescing after obtaining and scheduling cache-fit partitions, we show the performance of *memory coalescing only* (Org+R) versus our technique + *memory coalescing* for fair comparison and for demonstrating the significant performance improvement from our technique.

Among the other methods shown, first is CF, the Cache-Fit method described in Section 3.1. This splits the kernel to run each of the cache-fit partitions in standalone mode. Second is SJ, the Split-Join method described in Section 3.3 and uses tree-based set cover algorithm to merge cache-fit partitions. Third is CF-Q from Section 3.2, which applies the concurrent queue for loosely enforcing cache-fit partition ordering within a single kernel invocation. Finally we show SJ-Q from Section 3.4, which applies the concurrent queue to merged partitions in SJ. We use our Coarsened Bisection partitioner for all four of these methods. The baseline is the original program performance.

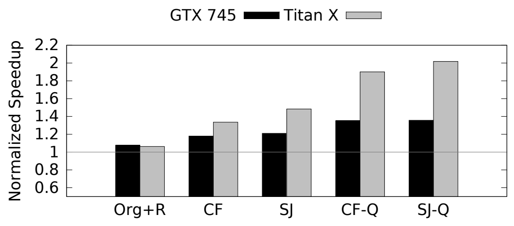
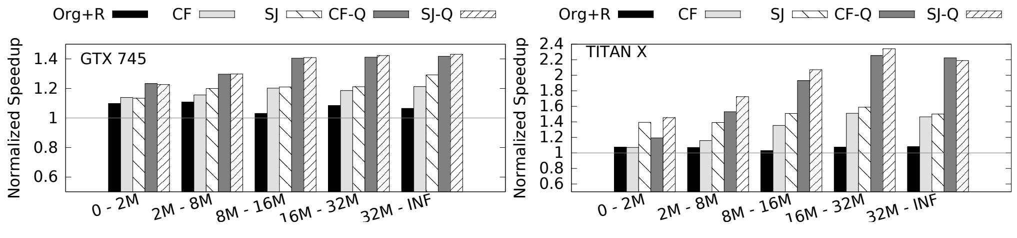


Figure 9: Average Speedup for SpMV

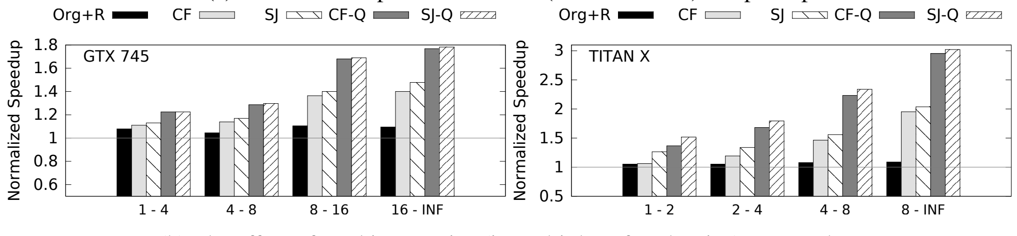
All four methods provide significant speedup compared to the original case and the Org+R case. But each method has trade-offs. The SJ and SJ-Q methods both require runtime profiling, whereas CF and CF-Q do not need runtime profiling. CF and SJ are easier to incorporate to a program as the kernel code does not change (only the input to each kernel invocation changes), whereas CF-Q and SJ-Q require code modification in order to implement the queue.

We find that our techniques provide significantly more improvement in the high contention environments of larger matrices with lower hit rates. We demonstrate the effectiveness of these methods with respect to matrix size, working set size, cache hit rate, and original running time.

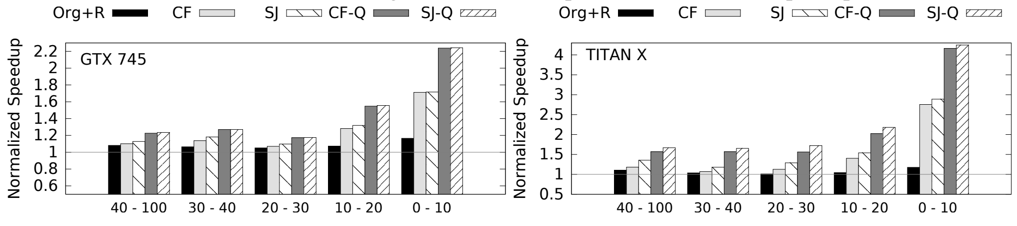
Matrix Size In Figure 10 (a), each group of bars shows average speedup for sparse matrices with the specified amounts of non-zeros. Every method except Org+R is



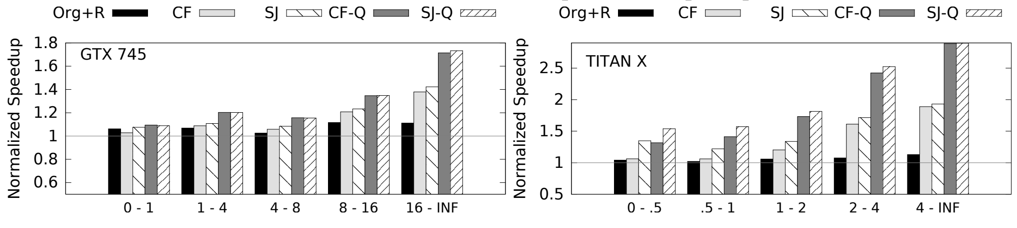
(a) The effect of sparse matrix size (# of non-zeros) on speedup



(b) The effect of working set size (in multiples of cache size) on speedup.



(c) The effect of cache hit rate (percent) on speedup.



(d) The effect of original runtime (milliseconds) on speedup.

Figure 10: SpMV Speedup on GTX 745 and Titan X

much more effective on larger matrices than on smaller ones, but we do see speedup in every group.

Working Set Our techniques become more effective as the working set grows, alleviating the increased cache contention. In Figure 10 (b), each group of bars shows average speedup for matrices with a working set of specified size. The unit used for the x-axis is the number of times the working set can completely fill the cache.

We see speedup improve as the working set grows, just as it tends to do when the matrix size grows. But the effects are more pronounced, with higher speedup. This shows working set size is more useful than matrix size for determining whether we should use locality-aware software throttling optimization.

Cache Hit Rate Our techniques are designed to improve matrices that suffer from low hit rates due to cache thrashing. As such, a lower original hit rate allows us to achieve higher speedup. In Figure 10 (c), each group of bars shows average speedup for matrices with a specified range of cache hit rates for the original case.

For matrices with lowest hit rates, the speedup for the queue-based approaches is particularly extreme. This shows that the queue-based approach is especially effective in environments that have high cache contention. The implicit communication between thread warps competing for queue reservations allows warps to achieve higher temporal locality with each other.

Run Time In Figure 10 (d), each group of bars shows the average speedup for matrices with the specified original runtime, measured in milliseconds. Since the Titan X device is much faster than the GTX 745, we use smaller thresholds for it. In general we can expect that the runtime correlates highly with the number of non-zeros, and so this figure shows a similar curve to the others.

CG We show the performance of conjugate gradient (CG) using SJ-Q. The major computation component is sparse matrix vector multiplication (SpMV). It calls *SpMV* iteratively until convergence. Therefore the overhead is amortized across different iterations. We show the overall performance in Figure 11 for a representative set of inputs. We find the performance improvement of CG with overhead is similar to that of *SpMV* without overhead. The overhead of Coarsened Bisection and SJ-Q profiling is well amortized.

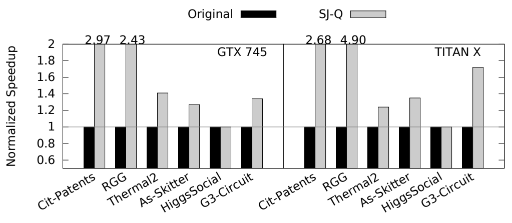


Figure 11: CG Speedup on GTX 745 and Titan X

We also show the L2 cache hit rates for CG in Figure 12. The changes to cache hit rates correlate with the performance improvement. The matrix rgg_n_2_23_s0 (RGG) has a much smaller cache hit rate on Titan X (0.44%) than on GTX 745 (36.75%), despite its larger cache size. There is more cache contention on Titan X since it uses more cores. We are able to improve the hit rate to 62.92% without changing the thread number or the implementation of the kernel code. Only the set of non-zero elements processed by each kernel is changed.

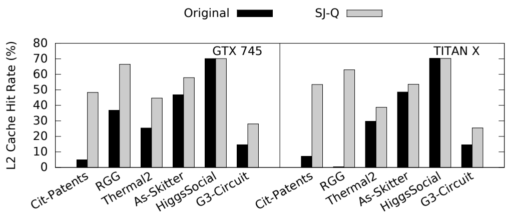


Figure 12: CG Cache Hit Rate on GTX 745 and Titan X

5.2 Neural Networks

We explore the effectiveness of our techniques on the AlexNet neural network, achieving an overall speedup of up to 54% on the Titan X device, which is suited for deep learning. Each fully connected layer of the neural network AlexNet operates as a matrix-vector operation; the matrix is a weight matrix. The work by Han and others [15] prunes the AlexNet network to remove elements of low weight and result in sparse matrices.

Since AlexNet is designed for smaller, embedded devices, we run multiple instances in parallel, allowing the neural network to analyze 150 different images at once for our Titan X GPU. This provides a reasonable amount of computation and data for our more powerful hardware.

In Figure 13, we show the speedup achieved by our technique on each of the three pruned *fc* layers of AlexNet. We include the alternate baseline of the original case plus data reorganization, as well as the Cache-Fit and Split-Join strategies both with and without the queue-based implementation.

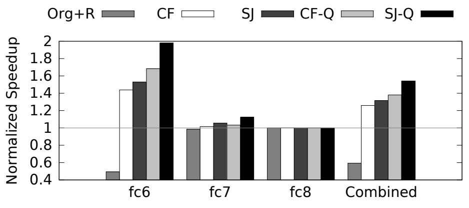


Figure 13: Speedup for AlexNet layers on Titan X

When only applying data reorganization, we see no improvement or even some slowdown. But when we apply any of our partitioning techniques we see speedup up to 98%, and no degradation on the smaller layers. The reason for less-speedup in the smaller layers (fc 7 and fc 8) is that their vector size is smaller and can fit into last level cache entirely in our Titan X GPU. We believe the performance improvement will be more pronounced for Alexnet if we test with embedded devices.

5.3 Graph Applications

We show the performance of the Bellman-Ford and PageRank programs on a set of graphs from the University of Florida Sparse Matrix Collection [11], Stanford Large Network Dataset Collection [23], and DIMACS implementation challenge [1]. Information for each graph is listed in Table 3.

We demonstrate the efficiency of the K-D tiling (SJ-kdtiling) approach, since both BMF and PR take fewer iterations to converge compared with sparse linear system solvers. Thus we need a fast and approximate partitioner so that the overhead can be amortized. We use SJ rather than SJ-Q, since it still provides good speedup while avoiding the overhead of the queue.

We summarize the performance with overhead in Table 3. We see that our approach improves performance for both BMF and PR. *RoadCal* benefits least, due to small size, but sees improvement in some cases.

Table 3: BMF and PR Performance Summary

Graph	GTX 745		TITAN X	
	BMF	PR	BMF	PR
Pokec [23]	1.62	2.98	1.88	3.17
WebGoogle [23]	2	3.29	1.8	3.37
Wikipedia-051105 [11]	1.24	1.99	1.43	1.97
WikiTalk [11]	1.74	2.57	2.09	2.75
IMDB [11]	2.16	3.22	1.59	2.62
RoadCentral [11]	1.19	1.6	1.69	2.18
RoadCal [1]	1	1	1	1.22

We observe that the speedup for PR is greater than for BMF. There are more memory accesses in the PR algorithm than in the BMF algorithm, and so it benefits more from our locality-aware software throttling.

We show cache hit rates for each program in Figure 14 and Figure 15. We show speedup with and without overhead for PR on Titan X in Table 4. PR has fewer iterations than CG so cannot improve performance with Coarsened Bisection if overhead is considered. However, the KD-Tiling method is fast enough that for SJ-kdtiling's performance to remain high with overhead.

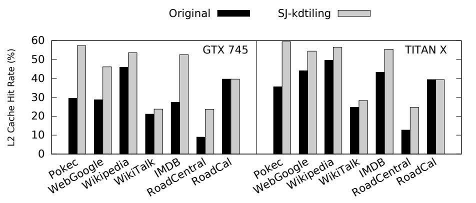


Figure 14: Cache Hit Rates for Bellman-Ford

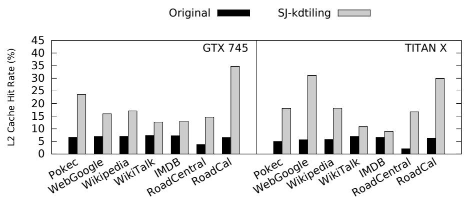


Figure 15: Cache Hit Rates for PageRank

Table 4: PageRank Speedup with and without Overhead

Graph	SJ-kdtiling w/ Overhead	SJ-kdtiling w/o Overhead
Pokec	3.17	3.34
WebGoogle	3.37	3.43
Wikipedia	1.97	1.98
WikiTalk	2.75	2.81
IMDB	2.62	2.27
RoadCentral	2.18	2.48
RoadCal	1.22	1.21

5.4 Computational Fluid Dynamics

The graph structure for CFD is a mesh in which every node has up to four neighbors. Since these meshes are small, We use SJ instead of SJ-Q for throttling. In Figure 16 we show the performance on GTX 745. We achieve speedup of up to 10%. Input *fvcorr_097* has the smallest number of nodes, thus the smallest improvement. CFD already has an optimized data layout [9]. With our throttling method, we nonetheless see some speedup. This demonstrates the effectiveness of our approach.

6 Related Work

Modern GPUs are equipped with massive amounts of parallelism and significant computing horsepower. However, this also results in higher levels of cache contention. To achieve high performance, reusing data in cache is critical. Both software and hardware approaches have been proposed to address the cache contention problem.

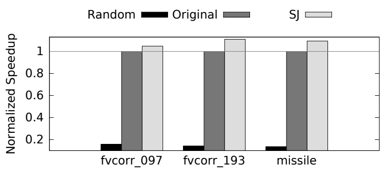


Figure 16: CFD Speedup

Warp Scheduling Policy Recent works focus on modifying GPU warp scheduling policy to reduce cache contention by throttling threads [18] [27] [20, 19] or to prioritize thread execution based on criticality [22]. However, all those approaches require hardware modification and require fine-grained thread scheduling which is complicated in a massively parallel system.

Our approach does not require hardware modification or fine-grained thread scheduling. Moreover, most warp scheduling policies aim to reduce the number of active warps for better performance. However, as we discovered in this paper, it is not always good to reduce the number of simultaneously running tasks for better cache performance. In some scenarios, i.e., when data reuse is low, having higher concurrency actually helps.

Computation and Data Layout Transformation On GPUs, Baskaran et al. [3] developed a compile-time transformation scheme coalescing loop nest accesses to achieve efficient global memory access. Zhang et al. [32] focused on reducing irregular memory accesses and enhancing memory coalescing to improve GPU program performance. These and other works [28, 31, 8, 17, 30, 4] all focus on improving memory coalescing for spatial locality. Our method is orthogonal to these approaches, as we optimize temporal locality.

7 Conclusion

This paper proposes a locality-aware software throttling framework that targets irregular sparse matrix applications on GPUs. We perform data-balanced *work partition* on the entire workload to get **cache-fit** partitions and use scheduling to exploit the trade-off between cache locality and execution pipeline utilization. Our framework is practical and effective. It requires no hardware modification and achieves an average 2.01X (maximal 6.45X) speedup on more than 200 real sparse matrices.

Acknowledgement We thank Peter Varman for being our shepherd and the anonymous reviewers for their constructive comments. This work is supported by grants NSF-CCF-1421505 and NSF-CCF-1628401. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] Dimacs implementation challenge - shortest paths, July 2013.

[2] Akbudak, K., Kayaaslan, E., and Aykanat, C. Hypergraph partitioning based models and methods for exploiting cache locality in sparse matrix-vector multiplication. *SIAM J. Scientific Computing 35*, 3 (2013).

[3] Baskaran, M. M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., and Sadayappan, P. A compiler framework for optimization of affine loop nests for gpgpus. In *Proceedings of the 22nd Annual International Conference on Supercomputing* (New York, NY, USA, 2008), ICS '08, ACM, pp. 225–234.

[4] Bell, N., and Garland, M. Efficient sparse matrix-vector multiplication on cuda. Tech. rep., Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[5] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517.

[6] Bourse, F., Lelarge, M., and Vojnovic, M. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2014), KDD '14, ACM, pp. 1456–1465.

[7] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)* (Washington, DC, USA, 2009), IISWC '09, IEEE Computer Society, pp. 44–54.

[8] Chen, X., Chang, L.-W., Rodrigues, C. I., Lv, J., Wang, Z., and Hwu, W.-M. Adaptive cache management for energy-efficient gpu computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2014), MICRO-47, IEEE Computer Society, pp. 343–355.

[9] Corrigan, A., Camelli, F., Löhner, R., and Wallin, J. Running unstructured grid cfd solvers on modern graphics hardware. In *19th AIAA Computational Fluid Dynamics Conference* (June 2009), no. AIAA 2009-4001.

[10] Dalton, S., and Bell, N. CUSP: A C++ templated sparse matrix library, 2014.

[11] Davis, T. A., and Hu, Y. The university of florida sparse matrix collection. *ACM Trans. Math. Softw. 38*, 1 (Dec. 2011), 1:1–1:25.

[12] Ding, C., and Kennedy, K. 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* (New York, NY, USA, 1999), PLDI '99, ACM, pp. 229–241.

[13] Fredman, M. L. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing 5*, 1 (1976), 83–89.

[14] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI* (2012), vol. 12, p. 2.

[15] Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR)* (2016).

[16] Hoare, C. A. Algorithm 65: find. *Communications of the ACM 4*, 7 (1961), 321–322.

[17] Jang, B., Schaa, D., Mistry, P., and Kaeli, D. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Trans. Parallel Distrib. Syst. 22*, 1 (Jan. 2011), 105–118.

[18] Jog, A., Kayiran, O., Chidambaram Nachiappan, N., Mishra, A. K., Kandemir, M. T., Mutlu, O., Iyer, R., and Das, C. R. Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 395–406.

[19] Kayiran, O., Jog, A., Kandemir, M. T., and Das, C. R. Neither more nor less: Optimizing thread-level parallelism for gpgpus. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques* (Piscataway, NJ, USA, 2013), PACT '13, IEEE Press, pp. 157–166.

[20] Kayiran, O., Nachiappan, N. C., Jog, A., Ausavarungnirun, R., Kandemir, M. T., Loh, G. H., Mutlu, O., and Das, C. R. Managing gpu concurrency in heterogeneous architectures. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2014), MICRO-47, IEEE Computer Society, pp. 114–126.

[21] Khorasani, F., Vora, K., Gupta, R., and Bhuyan, L. N. Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing* (New York, NY, USA, 2014), HPDC '14, ACM, pp. 239–252.

[22] Lee, S.-Y., Arunkumar, A., and Wu, C.-J. Cawa: Coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (New York, NY, USA, 2015), ISCA '15, ACM, pp. 515–527.

[23] Leskovec, J., and Krevl, A. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[24] Li, A., Song, S. L., Liu, W., Liu, X., Kumar, A., and Corporaal, H. Locality-aware cta clustering for modern gpus. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2017), ASPLOS '17, ACM, pp. 297–311.

[25] Li, L., Geda, R., Hayes, A. B., Chen, Y., Chaudhari, P., Zhang, E. Z., and Szegedy, M. A simple yet effective balanced edge partition model for parallel computing. *Proc. ACM Meas. Anal. Comput. Syst. 1*, 1 (June 2017), 14:1–14:21.

[26] Li, L., Geda, R., Hayes, A. B., Chen, Y., Chaudhari, P., Zhang, E. Z., and Szegedy, M. A simple yet effective balanced edge partition model for parallel computing. In *Proceedings of the 2017 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2017), SIGMETRICS '17 Abstracts, ACM, pp. 6–6.

[27] Rogers, T. G., O'Connor, M., and Aamodt, T. M. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2012), MICRO-45, IEEE Computer Society, pp. 72–83.

[28] Sung, I.-J., Stratton, J. A., and Hwu, W.-M. W. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2010), PACT '10, ACM, pp. 513–522.

[29] TSOURAKAKIS, C., GKANTSIDIS, C., RADUNOVIC, B., AND VOJNOVIC, M. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining* (New York, NY, USA, 2014), WSDM '14, ACM, pp. 333–342.

[30] VUDUC, R. W., AND MOON, H.-J. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *International Conference on High Performance Computing and Communications* (2005), Springer, pp. 807–816.

[31] YANG, Y., XIANG, P., KONG, J., AND ZHOU, H. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 86–97.

[32] ZHANG, E. Z., JIANG, Y., GUO, Z., AND SHEN, X. Streamlining gpu applications on the fly: Thread divergence elimination through runtime thread-data remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing* (New York, NY, USA, 2010), ICS '10, ACM, pp. 115–126.

[33] ZHANG, E. Z., JIANG, Y., GUO, Z., TIAN, K., AND SHEN, X. 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* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 369–380.