A Simple Yet Effective Balanced Edge Partition Model for Parallel Computing

LINGDA LI[*], Brookhaven National Lab
ROBEL GEDA, Rutgers University
ARI B. HAYES, Rutgers University
YANHAO CHEN, Rutgers University
PRANAV CHAUDHARI, Rutgers University
EDDY Z. ZHANG, Rutgers University
MARIO SZEGEDY, Rutgers University

Graph **edge** partition models have recently become an appealing alternative to graph **vertex** partition models for distributed computing due to their flexibility in balancing loads and their performance in reducing communication cost [6, 16]. In this paper, we propose a simple yet effective graph edge partitioning algorithm. In practice, our algorithm provides good partition quality (and better than similar state-of-the-art edge partition approaches, at least for power-law graphs) while maintaining low partition overhead. In theory, previous work [6] showed that an approximation guarantee of $O(d_{max}\sqrt{\log n \log k})$ apply to the graphs with $m = \Omega(k^2)$ edges (k is the number of partitions). We further rigorously proved that this approximation guarantee hold for all graphs.

We show how our edge partition model can be applied to parallel computing. We draw our example from GPU program locality enhancement and demonstrate that the graph edge partition model does not only apply to distributed computing with many computer nodes, but also to parallel computing in a single computer node with a many-core processor.

CCS Concepts: • **Mathematics of computing** → **Graph algorithms**; • **Theory of computation** → **Parallel computing models**; • **Computing methodologies** → *Modeling and simulation*;

Additional Key Words and Phrases: Graph model; edge partition; GPU; data sharing; program locality

ACM Reference format:
Lingda Li, Robel Geda, Ari B. Hayes, Yanhao Chen, Pranav Chaudhari, Eddy Z. Zhang, and Mario Szegedy. 2017. A Simple Yet Effective Balanced Edge Partition Model for Parallel Computing. *Proc. ACM Meas. Anal. Comput. Syst.* 1, 1, Article 13 (June 2017), 21 pages.
DOI: http://dx.doi.org/10.1145/3084451

1 INTRODUCTION

Graph **edge** partition models have recently become an appealing alternative to graph **vertex** partition models for distributed computing due to their flexibility in balancing workloads and their performance in reducing communication cost [6, 16]. In the edge partition model, a graph is partitioned based on the edges as opposed to vertices in the vertex partition problem. The edge partition model optimizes the number of vertex copies, which is a more precise measure for communication cost than the number of edge cuts.

[*]This work was done when Lingda Li was a Post-doc Associate at Rutgers University.

DOI: http://dx.doi.org/10.1145/3084451

We show an example of a computation problem abstracted by a graph edge partition model in Figure 1. We use the computation fluid dynamics (`cfd`) program [9, 10]. In `cfd`, the main computation is to calculate the interaction between fluid elements bounded by certain spatial distances and to use the interaction information to update the status of every fluid element for the next time step.

A fluid element is modeled as a vertex. An interaction is calculated between a pair of fluid elements, modeled as an edge. Figure 1 shows six interaction edges among six fluid elements. Figure 1(a) shows one possible way to

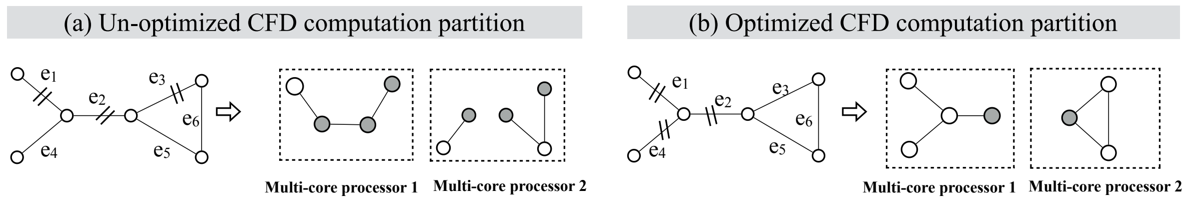


Fig. 1. Mapping of `cfd` interaction computation into two multi-core processors. Assuming we have two multi-core processors and each multi-core processor is in charge of three edges. In (a) there are three fluid elements that need to be copied into both processors while in (b) there is only one.

partition the computation (edges) to two multi-core processors, in which case, edges e_1, e_2, e_3 are mapped to one multi-core processor and e_4, e_5, e_6 are mapped to the other multi-core processor, requiring three fluid elements to be copied into both multi-core processors (marked as solid circles). However, with an optimized partition as illustrated in Figure 1(b), with e_1, e_2, e_4 mapped to one multi-core processor and e_3, e_5, e_6 mapped to the other multi-core processor, only one fluid element needs to be copied into both multi-core processors. In total, Figure 1(b) reduces the number of vertex copies by 66%, corresponding to a significant memory communication cost reduction.

Graph **edge** partition has been less studied in literature than graph **vertex** partition. The Powergraph [16] work is the first that discovered using edge partition rather than vertex partition improves work balancing and reduces communication cost in practice, especially for power-law graph. Bourse, Lelarge, and Vojnovic [6] established the first theoretical approximation guarantee and developed an efficient *streaming edge partition* algorithm. The target application for both studies is data analytics in large-scale distributed computing clusters, in which case good single-pass edge partition algorithms are desired.

The *hypergraph* partition approach [17] is an indirect approach to solve edge partition problem. The target application for *hypergraph* partition is parallel program that has potentially good data reuse and has multi-pass computation phases, for instance, sparse linear algebra solvers for optimization problems. In this paper, we target the same type of parallel application workloads as the *hypergrah* model.

The *streaming edge partition* algorithms [6, 16] are for scenarios when the input graph is given as an stream or only one pass through the input graph is allowed. In these settings their use is much preferred even when they may not yield the same partition quality as the *hypergraph* algorithm, their non-streaming counterpart. The reason is that the large overhead of the latter.

In this paper, we propose a novel edge partition approach that yields similar or improved partition quality than the *hypergraph* model, while at the same time, maintains acceptably low partition overhead. Our approach improves previous work in both theory and practice, in the following ways.

First, our approach is practical. It has significantly lower partition overhead than the *hypergraph* partition approach. Partitioning a power-law graph with 2 million vertices and 16 million edges takes about 400 seconds if we apply the best *hypergraph* algorithm we are aware of, while less than 18 seconds if we apply the approach proposed in this paper.

Second, it is simple yet effective. It is based on a transformation procedure called *split-and-connect* procedure we developed in this paper. The *split-and-connect* procedure improves the partition quality (the number of vertex copies in different partitions) over the *streaming* and the *hypergraph* approaches or is at least similar to the better of the two. In particular, when compared with the *streaming* edge partition algorithms, the partition quality improvement is the most pronounced in power-law graphs.

Third, it improves the theoretical guarantee for all ranges of parameters. Similar to the work by Bourse, Lelarge, and Vojnovic [6], by formulating the edge partition problem into vertex partition problem, we established an approximation guarantee of $O(d_{max}\sqrt{\log n \log k})$, where n is the number of vertices, m is the number of edges, k is the number of partitions, and d_{max} is the maximum degree. The work by Bourse, Lelarge and Vojnovic [6] showed that this approximation guarantee hold for graphs with the constraint $m = \Omega(k^2)$. We rigorously proved this approximation guarantee holds for all graphs.

We apply our edge partition approach to GPU programs and obtained non-trivial performance improvement. As far as we know, this is the first study of applying edge partition approach to GPU computing and to programs running on a many-core processor.

To summarize, our contributions are as follows:

- We propose a *split-and-connect* edge partition procedure that is easy to implement, yields good partition quality, and has low overhead.
- Our approach improves theoretical guarantee for all ranges of parameters.
- Our approach improves data locality in practice for massively parallel programs.

The rest of our paper is organized as follows. We present an abstract machine model for massively parallel programming in Section 2 to facilitate further discussion. Section 3 presents the *split-and-connect* partition procedure (Section 3.2), as well its application to real and synthetic graphs (Section 3.3), and proof for the approximation guarantee (3.4). Section 4 presents program transformations that are needed for applying the graph edge partition model to GPU programs. Section 5 shows our experimental environment and evaluation results. Section 6 discusses the related work, and Section 7 concludes the paper.

2 ABSTRACT MACHINE MODEL

We introduce an abstract machine model for GPU computing. We use GPU computing as an example here since it is a widely adopted massively parallel computing accelerator. Nonetheless, the abstract machine model applies to other parallel computing platforms as well.

We first describe the software cache model in GPU architecture. The software cache is a type of scratch-pad memory, that requires explicit read/write management. It is called *shared memory* using NVIDIA terminology[1]. In the rest of the paper, we use the term software cache and shared memory interchangeably for the same concept.

A GPU is composed of a set of streaming multiprocessors (SMs). The software cache is local cache for every SM, shared by threads running on the same SM. Multiple thread blocks run on one SM. Every thread block acquires a partition of the software cache, uses it, and yields it only when the thread block finishes its work. When one thread block yields the software cache partition, another thread block will claim the freed cache partition. During program execution, one thread block cannot peek into another thread block's software cache partition. It is as if every thread block has its own local cache and there are as many local caches as the number of logical thread blocks, despite the fact that the total physical space of software cache is typically limited. We show the abstract machine model in Figure 2, where every thread block can be viewed as a logical multi-core processor and every thread block is connected to a local cache.

The last level cache on GPU – the L2 cache is shared by all SMs on a GPU. An L2 cache on GPU has a high hit latency – typically 200 clock cycles and above, compared with the L1 hit latency which is typically less than 24

[1]We use NVIDIA CUDA terminology throughout the paper.

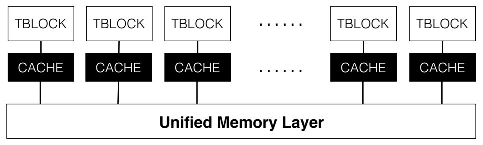


Fig. 2. Abstract GPU cache sharing model. TBLOCK refers to thread block.

clock cycles. L2 cache is shared by all SMs and is connected to the device DRAM. All DRAM accesses go through L2 cache. We abstract the L2 cache together with DRAM as an unified memory layer that is connected to all software local caches in Figure 2.

Hardware cache is similar to software cache. Only co-running threads share the hardware cache. There are several minor differences. Hardware cache automatically places/replaces data object and maintains address mapping. Cache replacement policies, such as least recently used (LRU) policy, are used to retain the more recently used data in cache. Thus, the hardware cache tends to keep the data for co-running threads present at the same time. L2 is hardware cache that is shared by all SMs. Other hardware caches, such as L1 cache, is local to every SM. However, they are typically used for special type of data objects, for instance, constant, read-only data objects, or thread-private memory [25] data objects. Thus L1 cache is not as extensively used as the software cache on GPUs. Thus we use the computing abstraction with software cache as an example for graph edge partition model.

3 DATA-CENTRIC LOCALITY MODEL

3.1 Problem Definition

With the abstract machine model defined in Section 2, we now describe how the graph edge partition model fits into the computing model. We also give a formal definition of the graph edge partition model.

The graph edge partition model places an emphasis on data. Computation is modeled as interaction between data. In the graph, a node represents a data object and an edge represents the interaction between two data objects.

```
neighbor_list * nlist =
       get_neighbour_list[cur_element];
for (i = 0; i < 3; i++) {
 neighbour = nlist[i];
  if ( neighbour >= 0 ) {
  density_nb = density[neighbour];
  movement_nb = movement[neighbour];
  speed_nb = speed[neighbour];
  pressure_nb = pressure[neighbour];
  compute_flux(density_nb, movement_nb,
               speed_nb, pressure_nb); }
}
```

Listing 1. CFD Pseudo Code for Interaction List Computation

We illustrate the graph edge partition model using a real program – computational fluid dynamic (cfd). We use the pseudo code of the `cfd` [8] program in Listing 1 to show its computation pattern. Every fluid element is modeled as a vertex and the interaction calculation *compute_flux* between a node and its neighbour is modeled as an edge. The computation is modeled as a list of interaction edges that need to be calculated. To parallelize the computation, one need to divide the edge list into different threads, ensure load balancing, and in the meantime, minimize communication cost. In our abstract machine model, the list needs to be divided evenly into different

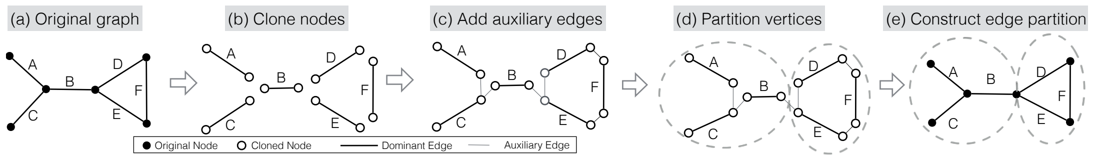


Fig. 3. Balanced edge partition problem converted to balanced vertex partition problem.

thread blocks, and the data needed by each thread block is copied into its local cache. We want to minimize the number of data copies in different local caches.

Formally, the balanced workload partition corresponds to the balanced edge partition. Our goal is to place the edges into k partitions, such that every edge is assigned to exactly one partition and the number of edges in every partition is the same.. The partition is named as a k-way balanced edge partition [6].

We follow the same problem formulation as the edge partition with aggregation (EPA) model defined in the work by Bourse, Lelarge, and Vojnovic [6]. The optimization problem is defined as follows:

Definition 3.1. Assume we have a graph $D = (V, E)$ with the set of vertices V and the set of edges $E \subset \binom{V}{2}$. Let n represent the number of vertices, m represent the number of edges, and k represent the number of partitions. Let L_i denote the number of edges in partition i. Assume x is a valid k-way balanced edge partition for D. $\forall v \in V$, assume v's incident edges are placed into p_v distinct partitions in x. As if the vertex v was copied into p_v partitions, we define the vertex copy cost for v as $p_v - 1$. We optimize the total vertex copy cost $C^{EP}(x)$:

$$\begin{array}{rl} \min & C^{EP}(x) = \sum_{v \in V}(p_v - 1) \\ \text{s.t.} & \forall i \in [1...k]\ L_i(x) = \frac{m}{k} \\ & x\ is\ a\ valid\ k-way\ edge\ partitioning \end{array} \tag{1}$$

The number of distinct partitions a node's incident edges fall into represents how often the corresponding data object is copied into different thread blocks' local caches. In the ideal case, one node's incident edges are placed in one and only one partition and thus there is no need to copy this node (data object) into more than one parition. One “copy" correspond to one memory load – loading the data object from off-chip memory to one local cache. We use p_v - 1 to measure how many copies (loads) are needed beyond the first copy (load) for every data object since the first copy is necessary – one node is placed in at least one partition. The total cost $C^{EP}(x) = \sum_{v \in V}(p_v - 1)$ the edge (work) evaluates the partition quality and also the memory load cost for a parallel program.

3.2 Partition Algorithm

The k-way balanced edge partition in **Definition 3.1** is NP-complete [6]. Compared with the balanced vertex partition problem, the edge partition problem is a much less studied and non-traditional graph partition problem. The most relevant studies we are aware of are the PowerGraph heuristic [16] and the weighted vertex partition approach by Bourse *et al.* [6]. The PowerGraph heuristics use a one-pass streaming algorithm that does not always yields good partition quality (as will be demonstrated in Section 3.3). The streaming algorithm by Bourse *et al.* [6] yields similar quality to the offline weighted vertex partition heuristic, however, it may not yield the best partition quality due to its emphasis on scalability (as will be discussed in Section 3.3).

An alternative approach to indirectly tackle the balanced edge partition problem is the hypergraph partition model [17], which has good partition quality but significantly larger overhead (as will be demonstrated in Section 3.3).

We propose an approach that is based on a graph transformation which we name as *split-and-connect* transformation. We denote the *split-and-connect* transformation as Ψ transformation. Our approach strikes a balance between efficiency and overhead, maintaining low partition overhead and good quality.

We sketch our partition algorithm in three steps:

(i) Given original graph G, we transform it into a new graph G' with the split-and-connect (Ψ) transformation.
(ii) We perform k-way balanced vertex partition on G' and obtain a valid partition y.
(iii) We construct a valid edge partition x for G using the vertex partition y for G', using the Ω transformation in Definition 3.3.

We describe every step in details together with a walk-through example in Figure 3.

① We convert the original graph using the split-and-connect transformation, defined as follows.

Definition 3.2. We define the split-and-connect graph transformation function Ψ. Assume $D = (V, E)$, $D' = (V', E')$ is a graph that is transformed from D using the Ψ transformation, $D' \in \Psi(D)$. In the split phase, for every vertex v of degree $d \in V$, we create a set of vertices $S(v) = \{\nu'_1, \dots \nu'_d\}$ in G', $S(v) \subset V'$.

In the connect phase, for any edge $e = (u, v)$ in G, we create a corresponding edge $e' = (\mu'_i, \nu'_j)$ in G', which we refer to as an **dominant edge**, such that $\mu'_i \in S(u)$ and $\nu'_j \in S(v)$, both μ'_i, ν'_j are connected to one and only one dominant edge e'. We then link the cloned vertices in every set $S(v)$ to form a path of d nodes and $d - 1$ edges, where d is the degree of v in G. We name such an edge as an **auxiliary** edge.

Since there are different ways to connect d nodes into one path, the Ψ transformation generates multiple different graph, and thus $\Psi(D)$ returns a set. One way to connect the d nodes from set $S(v)$ is to connect nodes based on their indices in the set: $\forall\ (\nu'_i, \nu'_{i+1})$ pair in set $S(v)$, $i = 1, \dots, d - 1$, we connect node ν'_i to node ν'_{i+1} with one auxiliary edge[2].

The split-and-connect transformation process is illustrated in Figure 3 (a), (b), and (c). In the transformed graph D', the total number of vertices is exactly twice the total number of edges in the original graph D, by construction.

② We partition the vertices of the transformed graph D'. We assign weight 1 to every auxiliary edge. We assign an infinite weight to every dominant edge so that during vertex partition of D', only auxiliary edges will be cut and no dominant edge will be cut. In practice we set it to 1,000 and it worked well (as demonstrated by experiment results in Section 5). Then we perform a k-way balanced vertex partition for D' and obtain a solution y such that only auxiliary edges are cut. The vertex partition process is illustrated in Figure 3 (d).

③ We reconstruct the edge partition solution x for D from the vertex partition solution y for D' obtained in the second step. Since in the solution y no dominant edges are cut, for every dominant edge, both end points fall into the vertex partition, assuming the i-th partition in y, and then we assign the dominant edge into the i-th partition in x. Every dominant edge in D' has a one-to-one mapping to an edge in D. Since the total number of vertices in every partition in D' is the same, and the number of dominant edges is half the number of vertices in the graph D', every edge partition in x has the same number of edges. By this step, we have successfully reconstructed the edge partition x for graph D. The process is formally defined below.

Definition 3.3. We define the edge partition construction function Ω. Assume a graph D' is transformed with the split-and-connect function Ψ, $D' \in \Psi(D)$. Let y be valid balanced k-way vertex partition of D' that does not cut any dominant edge. We construct an edge partition x of D form y of D' by the following.

Assuming $y = \{P_1, P_2, \dots, P_k\}$, where P_i is a disjoint partition of vertices in D'. For every dominant edge $e' = (u', v') \in D'$, find the vertex partition that u' and v' fall into in y. Assuming it is P_i; then we place the

[2]To simplify implementation, we also connect v_1 and v_d in the experiments for Figure 4, which does not change the theoretical approximation bound

Graph Feature				Hypergraph Partition Model				PowerGraph		WVP		SPAC	
matrices	# vertices	# edges	Default quality	hMETIS time (s)	hMETIS quality	PaToH time (s)	PaToH quality	Random quality	Greedy quality	Random quality	Greedy quality	time (s)	quality
cant	124902	2034917	53792	1344	51902	11.66	57050	3621269	281009	100921	76016	1.7	49708
circuit5M	11116652	59524291	22684508	NEM	N/A	2250	318709	85272145	16594380	19878734	10122128	67.2	285308
in-2004	2483424	16917053	3023386	NEM	N/A	413.6	132999	27057814	4211017	2044863	1154572	17.9	199881
mc2depi	1051650	2100225	144758	327.2	34789	4.87	35254	3118216	269827	39736	39029	1.44	36124
scircuit	341996	958936	200519	95.67	3997	2.91	4610	1502568	129838	10920	8608	0.64	5607

Fig. 4. Our proposed SPAC model vs. other partition methods. NEM represents not-enough-memory. Quality refers to the number of vertex copies in different partitions, as defined in Definition 3.1

corresponding edge $e \in D$ into the i-th edge partition of x. Then we finish the reconstruction process and $x = \Omega(D', y, D)$.

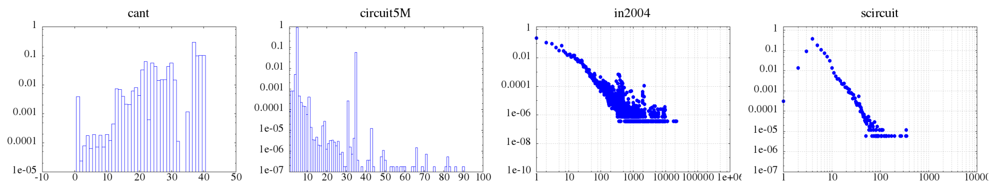


Fig. 5. Degree distribution. Note that y axis is log scale, and x axis is log scale for `in-2004` and `scircuit`.

The reconstruction process is illustrated in the example in Figure 3(d) and Figure 3(e).

3.3 Comparison with Existing Methods

We show how the split-and-connect (SPAC) partition approach works in practice by comparing it with existing approaches. We show the theoretical bound of the SPAC approach in Section 3.4.

For vertex partition component (step ii) of the transformed graph in the SPAC method, we use METIS [20] library, which is regarded the fastest vertex partition method in practice. We use representative graphs derived from sparse matrices in the Florida matrix collection [12] and matrix market [4], since graph structures are typically stored in sparse matrix format. We also use a graph generator to obtain synthesized graphs with different size, density, and degree distribution.

We describe the vertex degree distribution of the five real graphs first. The degree distribution of four of them are in Figure 5. The selected graphs have different degree distribution functions. The degree of `cant`'s graph is between 0 and 40. `circuit5M` has a more random degree distribution and we only show part of the x axis for readability. Two graphs exhibit power-law (i.e., scale free) pattern: `in-2004` and `scircuit`. We did not show `mc2depi` in Figure 5 since it has a relatively simpler degree distribution: more than 99% of vertices have degree 4, the rest have degree 2 or 3. The degree distribution of `mc2depi` resembles the mesh type graphs in scientific simulations.

3.3.1 *Comparison with hypergraph model.*

Hypergraphs are generalized graphs where an edge may connect more than two vertices, and such an edge is also called a *hyperedge*. In the hypergraph partition model [17], unlike our data-centric model, a vertex represents a task, and a hyperedge represents a data object where it covers all the vertices (tasks) that use this data object. The goal of minimizing data copies in caches is equivalent to minimization of hyperedge cuts when partitioning vertices (tasks) into k parts. Figure 6 shows an example on how to use the hypergraph model and comparison with the edge partition model. We also show the optimum

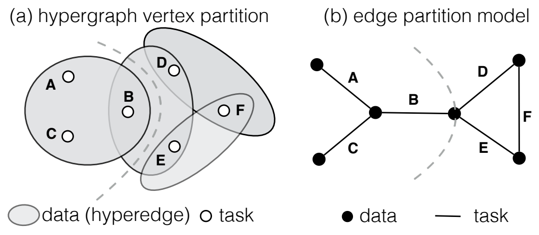


Fig. 6. Hypergraph model vs. edge partition model.

partition for both models in this example. In Figure 6(a) one hyperedge is cut, that corresponds to the one vertex cut in Figure 6(b).

For the hypergraph partition, we use hMETIS [21], a multilevel hypergraph partition tool, and PaToH [7], the fastest hypergraph partition implementation we are aware of.

From Figure 4, we see that PaToH is faster than hMETIS in the hypergraph model, and our basic SPAC model is significantly faster than both of them in all cases. The partition quality, measured as the data copy cost in Definition 3.1, shows that our SPAC model generates similar quality as PaToH and hMETIS. When ours is worse than of hyperpgraph model, for instance, in−2004 (where the gap is the largest), the ratio of the data copy cost beyond the ideal case over the data copy cost in the ideal case (in which every node is loaded once and only once from memory into cache) is low, ours is 8.0% while PaToH gives 5.3%, the difference between ours and hypergraph model is rather small. However, our model runs significantly faster than hypergraph models, between 4.5 times and 33.5 times faster.

Also note that the SPAC model has better scalability than the hypergraph model. For small graphs such as `scircuit`, our approach is 4 times faster, while for large graphs such as `circuit5M` and `in-2004`, our model is 20 or 30 times faster.

3.3.2 Comparison with PowerGraph heuristics. PowerGraph [16] proposes two simple edge partition approaches. Both methods scan all edges linearly to distribute them into partitions. The random method randomly assigns edges into every partition. The greedy method prioritizes the partitions that already cover one endpoint of the to-be-assigned edge. If no such partition is found, it chooses the partition with the fewest edges. Figure 4 shows the partition quality of both methods. Compared with hypergraph and our SPAC model, both PowerGraph heuristics have significantly worse partition quality because of the streaming nature.

3.3.3 Comparison with weighted vertex partition. Bourse *et al.* proposed a balanced edge partition approach based on weighted vertex partition (*WVP*) [6]. Every vertex is assigned a weight equivalent to its degree. Bourse *et al.* 's method partitions the degree-weighted graph into multiple balanced components with respect to total vertex weight.

Next, the edges that are not cut are assigned to the i-th edge partition corresponding if their end points fall into the i-th vertex partition. For edges that are cut, those whose endpoints fall into two different partitions, WVP assigns them either randomly or greedily to achieve a balanced partition. Bourse *et al.* also proposed a streaming online algorithm, since their online algorithm performs as well as their offline weighted-degree algorithms [6, 27], we compare with the offline algorithms here.

Figure 4 shows the partition quality of SPAC and two versions of WVP. The *WVP random* version assigns the cutting edges randomly to two partitions and the *WVP greedy* version uses a greedy assignment similar to PowerGraph's greedy partition. From Figure 4, we observe WVP greedy outperforms WVP random because of the blindness of random based method, and SPAC outperforms both WVP random and greedy.

We discover that the average degree of the nodes cut by SPAC is larger than that of the nodes cut by WVP. Table 1 shows the average degree of cutting vertices in SPAC and WVP greedy. Note that the average cutting degree is significantly larger when the partition quality improvement is larger, for instance, in circuit5M and in-2004. It is probably because the WVP model tends to avoid cutting large degree node more than the SPAC model. The WVP model performs weighted vertex partition on the original graph first. The goal of weighted vertex partition is to minimize the number of edges that are cut, and thus large degree nodes are less likely to be cut. However, this goal does not align with the ultimate goal of minimizing vertex copies, in which cutting a large degree node is potentially more beneficial. The intuition behind this is that cutting a large degree node can move more edges around for load balancing than cutting small degree nodes while in the meantime maintaining less data copy cost (the extent to which nodes appear in different partitions).

	WVP greedy	SPAC
cant	34.5	34.1
circuit5M	12.3	496.3
in-2004	57.9	159.6
mc2depi	4.0	4.0
scircuit	12.6	14.7

Table 1. Average degree of cutting vertices.

To gain more insights on the difference between SPAC and the WVP model, we implement a graph generator that generates graphs with different sizes, densities, and degree distributions. Figure 7 shows the partition quality improvement of SPAC over WVP greedy on generated graphs. We use two different types of graphs. The first is uniform distribution, in which every pair of nodes has the same probability being connected. The second one is power law distribution. We use the Barabasi–Albert model [2] to generate power law graphs. For both types of graphs, the number of vertices is kept the same, always 50k. The average degree increases, as shown on the x axis. The partition quality improvement is computed by dividing the quality of WVP greedy by that of SPAC. The results show that the benefit of SPAC over WVP greedy increases as the graph gets denser for both distributions. The results also show that our improvement is more pronounced when the maximum degree of the power-law graph increases. Although Figure 7 only shows the average degree on x-axis, the maximum degree increases as the average degree increases in the synthetic power-law graphs.

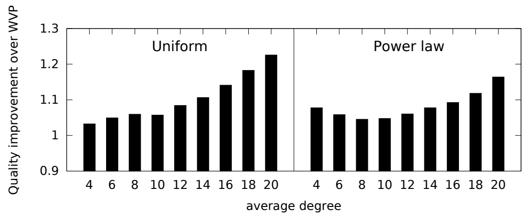


Fig. 7. Parition quality improvement compared with WVP greedy for synthesized graphs.

3.4 Analytical Bound

Our SPAC partition model is demonstrated to be effective with empirical evaluation. As far as we know, it also has the best provable approximation factor under the same load balancing constraints.

We prove that there exists a polynomial algorithm that approximates the optimal solution with a factor of $O(d_{max}\sqrt{\log n \log k})$, relying on the following facts:

Notations: Let $C^{VP}(y)$ be the cost of the balanced vertex partition y [15], which is the total number of edges whose end points fall into two different vertex partitions, also named as the number of edge cuts. Let $C^{EP}(x)$ be the cost of the balanced edge partition x (Definition 3.1 in Section 3.1).

Theorem 3.4. *Assume we have two graphs W and D such that $W \in \Psi(D)$, let y be a valid vertex partition for the graph W, let x be a valid edge partition of D, such that $x = \Omega(W, y, D)$, the edge partition cost of x is less than or equal to the vertex partition cost of y, $C^{EP}(x) \leq C^{VP}(y)$.*

Proof. When reconstructing x from W, y, and D using the Ω transformation in Definition 3.2, every auxiliary edge that is cut in y contributes at most one unit cost to the total cost of $C^{EP}(x)$ (the total number of node cuts for its incident edges to fall into multiple edge partitions). If l *auxiliary* edges in W that represent the same node v in D are cut, the node v is cut into at most $l+1$ distinct edge partitions in x, which contributes to l unit cost for $C^{EP}(x)$. If a dominant edge e is cut in y, at most one end point u of the dominant edge e get an additional unit cost for u potentially being cut into a different edge partition of x, since we place this dominant edge into one and only one edge partition in x. Only the edges that are cut in y will contribute to the edge partition cost of x. Thus $C^{EP}(x) \leq C^{VP}(y)$. □

Theorem 3.5. *For any graph D, there exist a graph W such that $W \in \Psi(D)$, W has an optimal vertex partition t_{opt} with the cost of $C^{VP}(t_{opt})$ that is equivalent to the cost of the optimal edge partition x_{opt} for D, $C^{VP}(t_{opt}) = C^{EP}(x_{opt})$. We refer to this graph W as the* **dual graph** *for D.*

Proof. Theorem 3.5 implies that the cost of the optimal edge partition for graph D is equivalent to the cost of the optimal vertex partition for graph W, where W is a graph constructed from D using the split-and-connect transformation Ψ.

To show W exist, we construct W from D by the following steps. Assume we have an optimal edge partition solution x_{opt} of D, we construct W using D and x_{opt}. Given a node v of degree d in D , we perform the split phase of the split-and-connect transformation as described in Definition 3.2 and create a set of d nodes $S = \{v'_1, v'_2, v'_3, ..., v'_d\}$ in W, each node v'_j corresponds to an incident edge e_j ($j = 1..d$) of v in D. In the connect phase, for the set S, we partition it into no more than k subsets S_i, $i = 1...k$ (assuming k is the number of edge partitions in D) such that S_i includes all the nodes $v'_{i_1}, v'_{i_2}, v'_{i_3}, .., v'_{i_q}$ that correspond to the edges in the i-th partition of x_{opt}, that is, $e_{i_1}, e_{i_2}, e_{i_3}, .., e_{i_q}$ belong to the i-th edge partition of x_{opt}. For every non-empty set S_i, we connect its nodes to form a path P_i. We connect P_1 to P_k to form a longer path (P_1 to P_2, P_2 to P_k and so on). The edges we use to create paths within and between S_i are *a*uxiliary edges.

We let the weight of every auxiliary edge be 1, the weight of every dominant edge be the total number of auxiliary edges. It ensures that the optimal vertex partition of W cuts only auxiliary edge.

Assume we have an optimal vertex partition solution t_{opt} for the graph W. We construct an edge partition solution x of the graph D using the function $x = \Omega(W, t_{opt}, D)$ (Definition 3.2). Using Theorem 3.4, the cost of the partition x is less than or equal to the cost of the partition t_{opt}: $C^{EP}(x) \leq C^{VP}(t_{opt})$.

It is obvious that the optimal edge partition x_{opt} of D can also be mapped to a valid vertex partition t of W by following the above construction process of W. The vertex partition t is obtained by cutting the edges that connect different S_i sets. Note that $C^{VP}(t) = C^{EP}(x_{opt})$ using this construction for W. We prove the solution t has minimal cost among all valid vertex partition solutions for W by contradiction. Assume there exists such a vertex partition μ of W that has smaller cost (fewer auxiliary edges are cut) than t of W, $C^{VP}(\mu) < C^{VP}(t)$, we use the reconstruction function Ω to convert it to an edge partition $\nu = \Omega(W, \mu, D)$ and since $C^{EP}(\nu) \leq C^{VP}(\mu) < C^{VP}(t) = C^{EP}(x_{opt})$, this contradicts that x_{opt} is optimal for D. Thus $C^{VP}(t) = C^{VP}(t_{opt})$, and we have $C^{VP}(t_{opt}) = C^{EP}(x_{opt})$. Therefore Theorem 3.5 is proved. □

Theorem 3.6. *$\forall M \in \Psi(D)$, assume the graph W is the dual graph of D, let y_{opt} be the optimal vertex partition of M, t_{opt} be the optimal vertex partition of W, and d_{max} be the maximal node degree of graph D, the following relation holds:*

$$C^{VP}(y_{opt}) \leq (d_{max} - 1) C^{VP}(t_{opt})$$

Proof. Since we do not know *a priori* the optimal edge partition x_{opt} of D, we do not immediately obtain the dual graph W of D as described in the proof of Theorem 3.5. However, we can determine an approximation factor of the optimal partition cost of an arbitrary split-and-connect transformed graph M with respect to the optimal vertex partition cost of the dual graph W.

For any $M \in \Psi(D)$, when connecting the cloned vertices $v_1', v_2', .., v_d'$ of one node v into a path, the order of the nodes that appear on the path from one end to the other is arbitrary. However the set of cloned nodes and the set of dominant edges every cloned node is associated with are the same, i.e., the set of cloned nodes for M and W is the same. Since vertex partition t_{opt} of W can be converted to a valid vertex partition s of the graph M by cutting any auxiliary edge of M if its two incident end points are in different vertex partitions of t_opt. For an arbitrary graph $M \in \Psi(D)$, if a path of auxiliary edges (corresponding to node v in D) is cut, at most $d_{max} - 1$ auxiliary edges are cut since the number of edges on the path is equivalent to the degree of the node v. Correspondingly, for any path that is cut in t_{opt} of W, the number of auxiliary edges cut per path is at least one. When converting t_{opt} to s, only if a path p_i (for the i-th node v_i in D) is cut in t_{opt}, the corresponding path p_i' (also for the i-th node v_i in D) will be cut in s. Therefore the vertex partition cost of s for M is at most $(d_{max} - 1)$ times of the vertex partition cost of t_{opt} for W, $C^{VP}(s) \leq (d_{max} - 1) * C^{VP}(t_{opt})$. Since y_{opt} is the optimal solution for M, $C^{VP}(y_{opt}) \leq C^{VP}(s)$, Theorem 3.6 is proved. □

Corollary 3.7. *For any graph M that is constructed from D, $M \in \Psi(D)$, there exists a polynomial time vertex partition solution y of M such that, $C^{VP}(y) \leq O(\sqrt{\log n \log k}) * (d_{max} - 1) * C^{VP}(t_{opt})$, where W is the dual graph of D and t_{opt} is the optimal vertex partition of the graph W.*

Proof. This step can be trivially proved using the Theorem 1.1 in [22] and Theorem 3.6 we proved. Theorem 1.1 in [22] states that there exists a polynomial time algorithm for balanced vertex partition problem with approximation factor of $\sqrt{\log n \log k}$.

Assuming that this solution is y for M,

$$C^{VP}(y) \leq O(\sqrt{\log n \log k}) * C^{VP}(y_{opt}).$$

According to Theorem 3.6, $C^{VP}(y_{opt}) \leq (d_{max} - 1) C^{VP}(t_{opt})$, therefore Corollary 3.7 is proved. □

Putting it altogether, we prove that there exist a polynomial algorithm that has $O(d_{max}\sqrt{\log n \log k})$ approximation factor for the balanced edge partition problem using the following four steps. We use the following notation. (D) represents the original graph, W is the dual graph of D, M is a graph constructed from D, $M \in \Psi(D)$. Let x_{opt} be an optimal edge partition of the graph D, t_{opt} be an optimal vertex partition for the dual graph W. Let y be a valid balanced vertex partition of M that satisfies Corollary 3.7 and let y_{opt} be an optimal balanced vertex partition of M. Let x be a valid edge partition constructed from M, D, and y, $x = \Omega(M, y, D)$. We show that x is the partition solution that satisfies the $O(d_{max}\sqrt{\log n \log k})$ approximation factor.

$$\begin{aligned}
C^{EP}(x) &\leq C^{VP}(y) && Step\ (1)\\
&\leq O(\sqrt{\log n \log k}) * C^{VP}(y_{opt}) && Step\ (2)\\
&\leq O(\sqrt{\log n \log k}) * (d_{max} - 1) * C^{VP}(t_{opt}) && Step\ (3)\\
&\leq O(\sqrt{\log n \log k}) * (d_{max} - 1) * C^{EP}(x_{opt}) && Step\ (4)
\end{aligned}$$

We prove the approximation bound in the above four steps. Step (1) is directly obtained using Theorem 3.4. Step (2) and (3) are obtained using Theorem 3.6 and Corollary 3.7. Step (4) can be obtained using Theorem 3.5.

Our approximation guarantee improves the work by Bourse *et al.* [6]. We not only proposed an alternative way to prove the approxiamtion guarantee but also proved the approximation guarantee holds for all ranges of parameters. Bourse *et al.* has relaxation of load balancing constraint, such that the maximum load size for every partition is $L(i) \leq (1 + \epsilon + k^2/m)$*(m/k), where m is the number of edges, $L(i)$ is the number of edges in the i-th partition, and k is the number of partitions, while our load constraint is $L(i) \leq (1 + \epsilon)$*(m/k), without relaxing the load balancing constraint. The ϵ factor comes naturally from the vertex partition component of our algorithm, since the traditional balanced (k, ϵ) vertex partition problem allows a factor ϵ. In both our work and the work by Bourse, Lelarge, and Vojnovic, ϵ is set to 1.

4 LOCALITY ENHANCEMENT

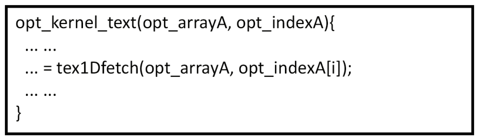
We describe how to apply the SPAC partition results to enhance GPU program locality.

Program Transformation The first step is to perform the job swapping and data reordering program transformations as described in [29]. The SPAC partition results determine how tasks should be scheduled to different thread blocks. Job swapping is a program transformation that enables task swapping among different threads. However, job swapping relies on an input schedule that guides the mapping of tasks to threads. The input schedule here is determined by our locality graph partition results. The key transformation technique for job swapping [29] is simple – replacing the logical thread id with a new threads id such that $tid' = newSchedule[tid]$ while the **newSchedule[]** array is determined by graph partition results and it is passed as an additional kernel argument. Since GPU programs use thread id to infer the task for every thread, transformation of thread id can change the task a thread. Similarly other thread index information such as thread block id also needs to be transformed if they are used in the kernel.

Another program transformation that follows job swapping is data reorganization if job swapping adversely affects memory coalescing efficiency. Data reorganization depends on thread scheduling information. We use existing technique to perform data layout transformation [29] [13]. The key idea is to reorganize data in memory based on the new schedule of tasks, for example, place data processed by threads in the same warp (or block) near each other in memory.

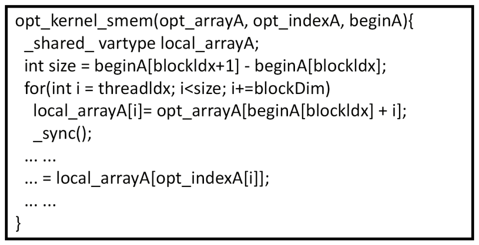
The second step is to pre-load data shared by threads in the same thread block into cache. If we use software cache, we need to explicitly pre-load data into cache and determine the addresses in software cache, as illustrated in Figure 8 (a). If we use hardware cache, for instance, the texture cache, we need to let the host function bind the particular data objects to texture memory using the CUDA built-in function *cudaBindTexture()*. The GPU kernel prefixes every texture cache data reference using *tex1Dfetch()* as shown in Figure 8(a).

Overhead Control To reduce runtime overhead, we can perform SPAC partitioning and data layout transformation using the CPU while kernel is executed on the GPU. The CPU-GPU pipelining technique is proposed in [28, 29] to overlap GPU computation and locality optimization so that the overhead is transparent. Our locality optimization is performed once and we reuse the task schedule and data layout for performance improvement for computation. We illustrate this overhead control model with an example in Figure 9. The CPU optimization



```
opt_kernel_text(opt_arrayA, opt_indexA){
  ... ...
  ... = tex1Dfetch(opt_arrayA, opt_indexA[i]);
  ... ...
}
```

(a) Kernel using texture cache



```
opt_kernel_smem(opt_arrayA, opt_indexA, beginA){
  _shared_ vartype local_arrayA;
  int size = beginA[blockIdx+1] - beginA[blockIdx];
  for(int i = threadIdx; i<size; i+=blockDim)
    local_arrayA[i]= opt_arrayA[beginA[blockIdx] + i];
    _sync();
  ... ...
  ... = local_arrayA[opt_indexA[i]];
  ... ...
}
```

(b) Kernel using software cache

Fig. 8. Example code transformation.

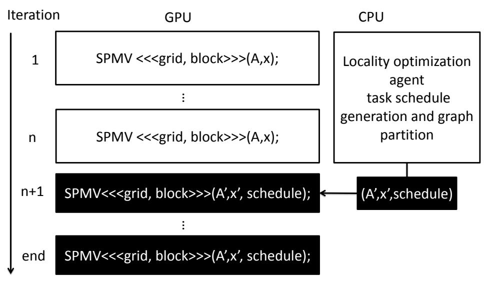

Iteration
GPU
CPU
1
SPMV <<<grid, block>>>(A,x);
n
SPMV <<<grid, block>>>(A,x);
Locality optimization agent
task schedule generation and graph partition
n+1
SPMV<<<grid, block>>>(A',x', schedule);
(A',x',schedule)
end
SPMV<<<grid, block>>>(A',x', schedule);

Fig. 9. Overhead Control.

agent is working in parallel with the first n iterations and once it is done, the locality-optimized task schedule and data layout is applied to the $n + 1$ iteration. We perform SPAC partition within the CPU optimization agent.

Rong *et al.* points out that in sparse linear algebra applications, once the best data layout and/or computation schedule is obtained, it can be reused in multiple loop iterations or for different computation task in the same loop. We want to point out that other applications (besides sparse linear algebra) also share this property, for instance, fluid dynamics and graph processing algorithms. Thus for this important set of applications, the overhead can be amortized due to reuse of the partition result. We focus on this set of applications and describe in more details in Section 5.

In the CPU-GPU pipeline model, the master CPU thread checks if the optimization agent is completed at the end of every iteration and if so apply the optimization results. If the optimization thread does not complete when the program finishes, the master thread terminate it to ensure no performance degradation. We use this model to take the overhead of SPAC into consideration and present the results in Section 5.

5 EVALUATION

5.1 Experimental Methodology

We conduct the locality enhancement experiments on two GPUs: the NVIDIA Titan X (Pascal architecture) and the GeForce GTX 680 (Kepler architecture) to evaluate the performance of *SPAC*. Table 2 shows the configuration. There are three configurations of L1 cache and shared memory for GTX 680: 16KB/48KB, 32KB/32KB and 48KB/16KB. Since L1 cache is not used for global memory data for GTX 680, we always configure shared memory to be 48KB. We conduct the *SPAC* partition experiments on CPU – the Intel Core i7-4790 CPU with 4 cores at 3.6 GHz. Currently, the *SPAC* partition algorithm only utilizes one core, and thus the partition cost may be further reduced using if using multi-core .

GPU Model	Titan X	GTX 680
Architecture	Pascal	Kepler
SM #	28	8
Core #	3584, 128 per SM	1536, 192 per SM
Shared memory	96KB per SM	48KB per SM
Texture cache	24KB per SM	48KB per SM
Linux kernel	2.6.32	3.1.10
CUDA version	CUDA 8.0	CUDA 5.5

Table 2. Experimental environment.

Benchmark	Application Domain
b+tree [8]	Tree search
cfd [8]	Computation fluid dynamics
gaussian [8]	Gaussian elimination
particlefilter [8]	SMC for posterio density estimation
SPMV [11]	Sparse matrix vector mult.
CG [11]	Conjugate gradient solver

Table 3. Benchmark summary.

Our benchmarks are listed in Table 3. We use six applications from various computation domains as described in Table 3. This set of benchmarks are representative of important contemporary workloads that can benefit from cache locality enhancement.

We show the detailed experiment results for SPMV and CG in Section 5.2. We report the summary experiment results for other four benchmarks in Section 5.3.

5.2 Detailed Analysis Results

We present the detailed analysis results for SPMV and CG for several reasons. First, SPMV is an important computation kernel in many applications such as numerical analysis, PDE solvers, and scientific simulation. Second, the conjugate gradient (CG) [19] is an application calls SPMV iteratively, representing an application case of SPMV. Therefore, we can present both the performance improvement in individual kernel and the performance-overhead trade-off in a kernel to show how practical it is. We use real-world sparse matrices from the University of Florida sparse matrix collection [12] and matrix market [4] as input to CG.

We compare our approach with the highly optimized implementation in cuSPARSE [26] and CUSP [11] libraries. The CUSP SPMV kernel is open source. It reorders the data in a pre-processing step such that all non-zero elements are sorted by row indices and then it distributes the non-zero elements evenly to threads. We are not aware of cuSPARSE's SPMV method implementation since its source code is not disclosed. However since it is a popular and widely used library, and it is faster than CUSP for most of the times, we also include cuSPARSE for comparison.

Name	Dimension	Nnz	cuSPARSE time	SPAC time
cant	62K*62K	2.0M	0.74	0.81
circuit5M	5.6M*5.6M	59.5M	15,806.64	211.82
cop20k_A	121K*121K	1.4M	6.74	6.63
Ga41As41H72	268K*268K	9.4M	7.03	4.24
in-2004	1.4M*1.4M	16.9M	157.99	68.48
mac_econ_fwd500	207K*207K	1.3M	8.83	6.83
mc2depi	526K*526K	2.1M	10.72	10.95
scircuit	171K*171K	0.96M	8.87	5.04

Table 4. Matrix Information. Nnz represents the total number of nonzero elements. cuSPARSE and SPAC shows total SPMV kernel execution time in CG using the cuSPARSE, SPAC methods respectively on Titan X (in seconds).

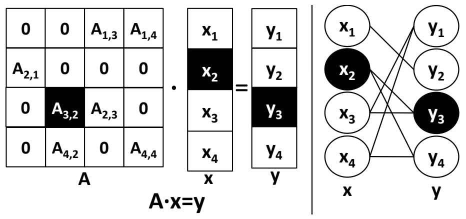


Fig. 10. Building locality graph for SPMV.

To construct a locality graph for SPMV, we let one vertex correspond to one element in the input vector x, and one vertex correspond to one element in the output vector y. For any non-zero element $A[i, j]$ in the input matrix A, an edge connects the vertex j in the input vector and the vertex i in the output vector, since a non-zero $A[i, j]$ implies a multiplication of $A[i, j]$ with x_j and an addition to y_i. We show an example on the locality graph for SPMV in Figure 10.

With the locality graph constructed, we perform edge partition using the *SPAC* approach and we let one thread block be mapped to one partition of the graph. We use software cache to store the necessary input and output vector elements within each thread block, that is, all the unique input and output vector elements used in every partition. Different thread blocks might have different shared memory demand since the number of unique data elements in every partition might be different. We pick the largest demand and set it as the shared memory (software cache) usage per block (since the shared memory size for each thread block needs to be the same). We will also show the results of using texture cache for only the input vector elements.

Table 4 shows the 8 matrices used in our experiments. We use matrices that have different data access patterns to show the applicability of *SPAC*. Note that here the execution time is the total SPMV kernel time in CG application and it does not take the partition overhead into consideration.

Figure 11 and 12 compare the performance of four SPMV versions on Titan X and GTX 680 respectively, including cuSPARSE, CUSP, the *SPAC* version that does not consider partition overhead (*SPAC*), and the version that takes overhead into consideration (*SPAC-adapt*). *SPAC-adapt* results are obtained using the overhead control model described in Section 4. We set the thread block size to 1024. We use cuSPARSE kernel time as the baseline, since it is faster than CUSP for most matrices.

We observe *SPAC* based approach is faster than both of cuSPARSE and CUSP in most cases, except *SPAC* has a marginal slowdown for `cant`. When using *SPAC-adapt* for `cant`, there is almost no slowdown compared with the baseline. *SPAC* is slightly worse than CUSP on GTX 680 for `in-2004` because the working set for some thread block is large, thus causing *SPAC* to use a large amount of software cache, adversely affecting occupancy. *SPAC* benefits from the large shared memory on Titan X, and thus it outperforms CUSP on Titan X for `in-2004`. We also observe that in most cases, the performance of *SPAC-adapt* is similar to that of *SPAC* except for `Ga41As41H72` and `cant`, where the partition time is large as compared with the kernel execution time and we can only optimize

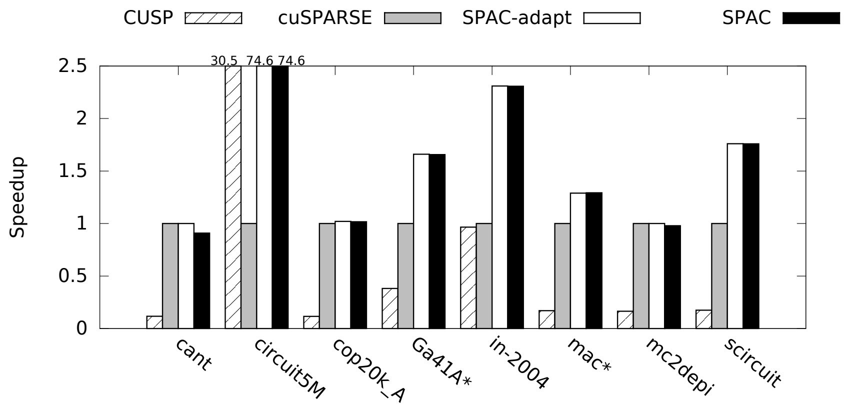


Fig. 11. SPMV performance on Titan X.

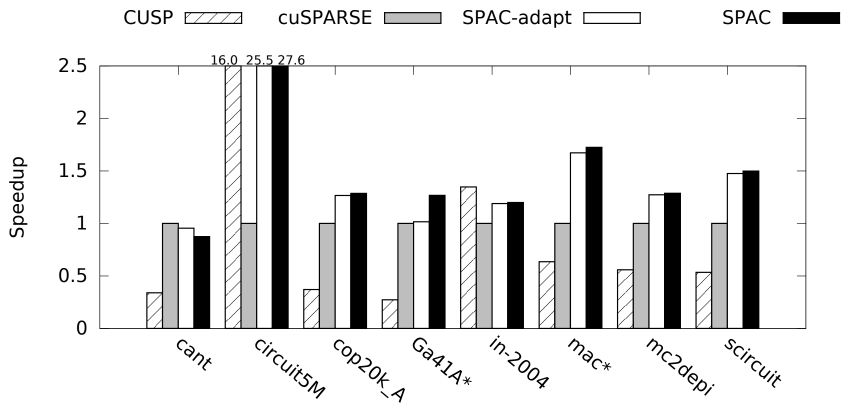


Fig. 12. SPMV performance on GTX 680.

a small portion of SPMV invocations. Overall, the performance results are similar on both GPUs, and thus we mainly show the results for Titan X in the rest of this section.

Figure 13 shows the normalized memory transaction number for three SPMV kernels on Titan X. All results are normalized to that of cuSPARSE. We observe that memory transaction number is reduced significantly for all matrices except cant and Ga41As41H72. Overall, the transaction reduction maps well to the performance results.

Next, we compare the performance of *SPAC* when using texture cache or shared memory for the input vector. Since the output vector is for reduction type writes, texture cache cannot be used to handle it. The shared memory (software cache) is still used to store the elements in the output vector in texture cache version. Table 5 compares their performance under different thread block size on Titan X. The texture and software cache versions are represented as tex and smem respectively. Software cache version outperforms texture cache version for almost all matrices although by a small percentage. Note that the texture cache version still outperforms CUSP and cuSPARSE in Figure 11 and 12.

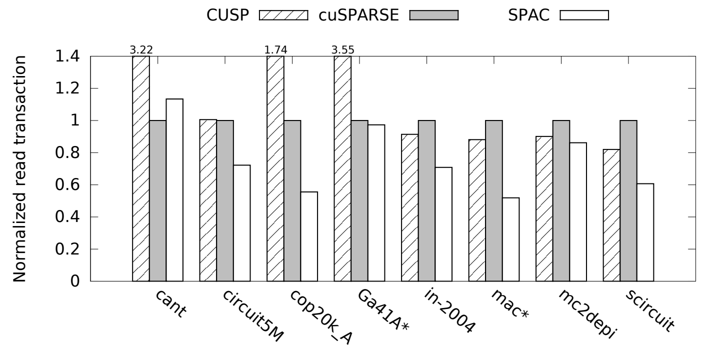


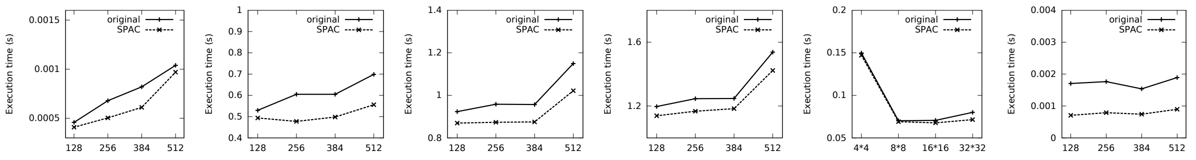
Fig. 13. Normalized transaction number for SPMV.

Block size	256		512		1024	
time (μs)	tex	smem	tex	smem	tex	smem
cant	114	109	116	112	119	114
circuit5M	3007	2843	3160	2976	2977	2798
cop20k_A	76	77	78	75	77	78
Ga41A*	450	432	478	453	469	437
in-2004	830	799	858	816	857	807
mac*	76	75	77	75	78	77
mc2depi	131	121	133	126	136	127
scircuit	59	58	59	59	61	60

Table 5. SPMV performance with different thread block sizes and cache types on Titan X.

In general, the texture cache based approach could potentially pollute cache by evicting data before it gets fully reused, while using software-managed cache does not have such a problem. When hardware cache have performed similarly well as the shared memory case, it may be be considered to help improve programmability.

Finally, we show the sensitivity of our approach with respect to different thread block sizes. Table 5 shows the execution time of one kernel invocation in *SPAC* under different thread block sizes. The results suggest the performance at different thread block sizes are similar to each other. However, using smaller block size implies larger block/partition number, and thus longer partition time of *SPAC*. Taking both kernel performance and partition overhead into consideration, we choose 1024 as the default block size in our experiments.



(a) b+tree (b) cfd (f097K) (c) cfd (f193K) (d) cfd (m0.2M) (e) gaussian (f) particlefilter

Fig. 14. Application performance on different block sizes.

5.3 General Analysis Results

Figure 14 shows the performance of four Rodinia applications under various thread block sizes on Titan X. The original execution time is denoted as *original* in Figure 14. The execution time of *SPAC* is denoted as *SPAC*. In these cases, the locality graphs are known at compile time or the SPAC partition information can be embedded into the input file format, i.e., the *cfd* benchmarks in Rodinia Benchmark suite. We show the experiments for before using *SPAC* approach and after using *SPAC* approach without taking the overhead into consideration.

First, we observe that in most cases our optimized version outperforms the original version. The maximum speedup is 2.40x for `particlefilter` at the thread block size 128.

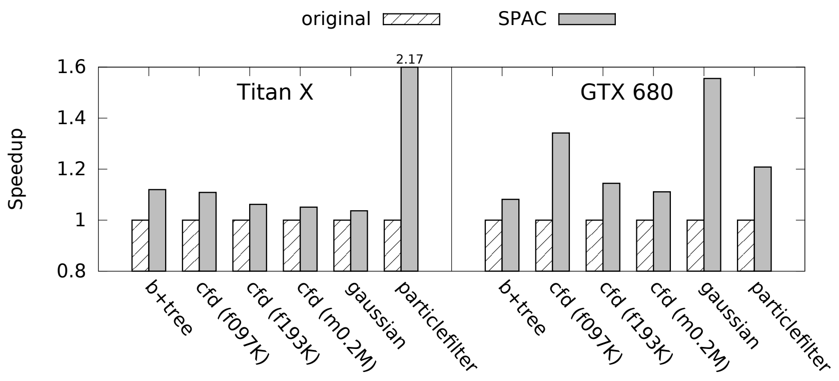


Fig. 15. General application performance summary.

Second, we observe that for most benchmarks, the *SPAC* approach improves performance at every thread block size, except for *gaussian*. For *gaussian*, we have the speedup only for large thread block sizes. It is potentially because with larger thread block size, there are more threads within a thread block and thus more potential data sharing. The benchmark *gaussian*'s first two thread block sizes (inherent in the program) are no more than 64, thus less data sharing potential. Thus the best speedup is obtained when the thread block size is 256 or larger.

Besides more potential data sharing, another advantage of larger thread block size is lower partition overhead since the partition number decreases. Nonetheless, for the best performance of one program, it is not always good to use larger thread block. For instance, there is a significant performance degradation from the thread block size 384 to 512 for all three inputs of `cfd` for the original program. This is because thread block size may affect other performance factors, for example, the occupancy. For `cfd`, at the thread block size of 384, Titan X can fit two thread blocks in each SM and achieve a total occupancy of 768 threads per SM, while only one thread block can run on one SM at the block size of 512.

To ensure fair comparison, we pick the best performance across all thread block sizes for both the original version and the *SPAC* version. In Figure 15, we show the best performance of SPAC over different thread block sizes versus the best performance of the original program across different thread block sizes on both Titan X and GTX 680. All data are normalized to the original version on that GPU. SPAC achieves significant performance gains, or at least no performance degradation, for all benchmarks. The performance improvement is similar for `b+tree` on both GPUs. For `particlefilter`, SPAC performs better on Titan X because they benefit from the larger shared memory on Titan X. For `cfd` and `gaussian`, SPAC achieves better performance improvement on GTX 680. For `gaussian`, the original kernel performs best at the block size of 8*8 on Titan X as shown in Figure 14e, but there is not much data sharing to exploit for SPAC at such a small block size. Although SPAC successfully exploits data sharing and achieves better performance at larger block size, at the block size of 8*8 it does not significantly outperform the original kernel.

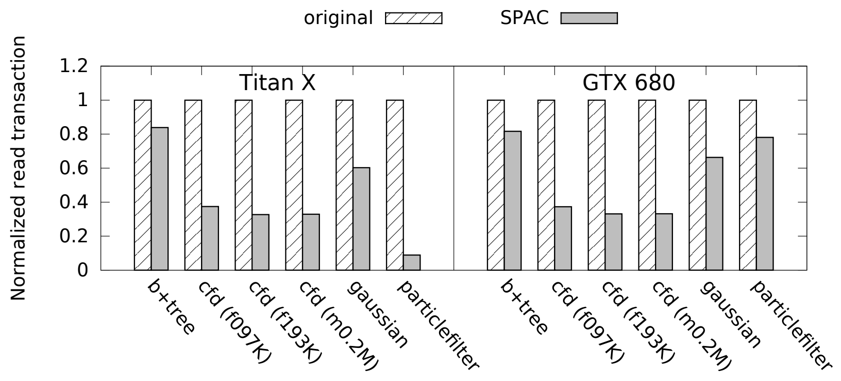


Fig. 16. Read transaction number.

Figure 16 shows the normalized L2 cache read transaction number measured by CUDA profiler for this set of benchmarks. The reason why we do not show write transaction number is that there is no write sharing in these benchmarks. All results are normalized to that of the original version at the same thread block size. The results show that our technique can reduce memory transactions significantly, since more memory requests are satisfied by software or texture caches within each SM. The performance results in Figure 15 are well correlated with the memory traffic results in Figure 16.

6 RELATED WORK

Graph Partition Model for Parallel and Distributed Computing

Graph partition models have been proposed to characterize data communication in parallel CPU computing systems, but most of them focus on graph vertex partition. Hendrickson *et al.* study vertex-partition based graph models in which vertices represent tasks [17, 18] and edges represent data communication. The vertex-partition model correlate edge cut cost with data communication cost but do not directly give communication cost.

Hypergraph models [7, 21] can model communication cost directly. Hypergraph partitioning is used for cache optimization. Ding *et al.* [14] show that optimal loop fusion for cache locality can be solved by k-way min-cut of a hypergraph, and the problem is NP-hard for k>2. This formulation is recently generalized in the work by Kristensen *et al.* [1]. A solution of k-way min-cut does not provide balanced partitioning and hence cannot be used to solve the problem addressed in this paper.

The main drawback of the hypergraph model is its large overhead, as we demonstrate in Section 3.3, which makes it infeasible for massively parallel computing.

Pregel [24] and GraphLab [23] introduce two parallel computation models based on message passing and shared memory computing model, respectively. They assign all computation of one vertex to one processor due to limitations of the graph partition model, while edge partition model allows computation of the same vertex to be scheduled to different processors to achieve better partition quality and load balance.

PowerGraph [16] uses graph edge partition and thus can distribute the computation of one vertex into several processors. It also proposed random and greedy edge partition methods, but mainly streaming algorithms. Bourse *et al.* propose to use weighted vertex partition to achieve balanced edge partition [6]. In Section 3.3, our experiments and analytical analysis demonstrate our approach improves partition quality and also improves the theoretical approximation guarantee.

Other Models for Improving Communication Cost in Parallel Computing

Bondhugula *et al.* introduce an automatic source-to-source transformation framework to optimize data locality, and they formulate data locality problem with polyhedral model [5], which only works for regular applications with affine memory indices. The graph model can be applied to these regular applications as well as irregular applications, however, the polyhedral model can't be applied to irregular applications, i.e., sparse linear algebra solvers.

Ding and Kennedy proposed to use runtime heuristic for improving memory performance of irregular programs. The programming support is known as the inspector-executor model. For locality optimization, this can be automated and optimized by a compiler, as done initially for sequential code [DingK:PLDI99] and later for parallel programs [Venkat+:PLDI15].

Most GPU dynamic workload partition studies mainly focus on optimizing memory coalescing rather than data reuse, which is spatial locality rather than temporal locality. Zhang *et al.* propose to dynamically reorganize data and thread layout to minimize irregular memory accesses [29]. Wu *et al.* also propose two data reorganization algorithms to reduce irregular accesses [28] for more coalesced memory accesses.

There are also domain-specific studies for improving the memory performance of sparse matrix vector multiplication. Bell and Garland discuss various sparse matrix representation formats [3]. Venkat and others propose transformations to automatically choose sparse matrix representation and optimize inspector performance in the inspector-executor model.

7 CONCLUSION

In this paper, we propose a simple yet effective graph edge partition technique based on the split-and-connect heuristic to improve data communication cost. In practice, our algorithm provides good partition quality (and better than similar state-of-the-art edge partition approaches, at least for power-law graphs) while maintaining low partition overhead. In theory, we improved previous work [6] by showing that an approximation guarantee of $O(d_{max}\sqrt{\log n \log k})$ hold for graphs with all ranges of parameters – that is, not just hold for the graphs with $m = \Omega(k^2)$ edges (k is the number of partitions).

This is also the first comprehensive study of how to characterize and optimize data communication cost using graph edge partition model for massively parallel GPU computing platform. Compared with previous graph partition models, our method provides high quality task schedule and yet has low-overhead. Our experiments show that our method can improve data sharing and thus performance significantly for various GPU applications.

ACKNOWLEDGEMENTS

We thank Milan Vojnovic for providing invaluable comments during the preparation of the final version of the paper. We owe a great debt to the anonymous reviewers for their helpful suggestions on this paper. This work is supported by NSF Grant NSF-CCF-1421505, NSF-CCF-1628401, and Google Faculty Award. 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] Fusion of parallel array operations. pages 71–85, 2016.
[2] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.
[3] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.
[4] R. F. Boisvert, R. Pozo, K. A. Remington, R. F. Barrett, and J. Dongarra. Matrix market: a web resource for test matrix collections. In *Quality of Numerical Software*, pages 125–137, 1996.

[5] 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.

[6] 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.

[7] U. V. Catalyürek and C. Aykanat. PaToH: a multilevel hypergraph partitioning tool, version 3.0. *Bilkent University, Department of Computer Engineering, Ankara*, 6533, 1999.

[8] 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.

[9] A. Corrigan, F. Camelli, R. Löhner, and J. Wallin. Running unstructured grid cfd solvers on modern graphics hardware. In *19th AIAA Computational Fluid Dynamics Conference*, number AIAA 2009-4001, June 2009.

[10] A. Corrigan, F. Camelli, R. LÃűhner, and J. Wallin. Running unstructured grid-based cfd solvers on modern graphics hardware. *Int. J. Numer. Meth. Fluids*, 66:221–229, 2011.

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

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

[13] 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.

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

[15] G. Even. Fast approximate graph partitioning algorithms. *SIAM J. Comput.*, 28(6):2187–2214, Aug. 1999.

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

[17] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Comput.*, 26(12):1519–1534, Nov. 2000.

[18] B. Hendrickson and T. G. Kolda. Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing. *SIAM Journal on Scientific Computing*, 21(6):2048–2072, 2000.

[19] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. 1952.

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

[21] G. Karypis and V. Kumar. hMETIS 1.5: A hypergraph partitioning package. Technical report, Department of Computer Science, University of Minnesota, 1998.

[22] 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.

[23] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[24] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[25] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. 2012.

[26] NVIDIA. cuSPARSE library. 2014.

[27] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, pages 333–342, New York, NY, USA, 2014. ACM.

[28] 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.

[29] 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.

February 2017

Received November 2016; revised accepted