Parallel Graph Partitioning on a CPU-GPU Architecture

Bahareh Goodarzi
Department of Computer Science
Concordia University
Montreal, Quebec, Canada
b_goodar@encs.concordia.ca

Martin Burtscher
Department of Computer Science
Texas State University
San Marcos, TX, USA
burtscher@txstate.edu

Dhrubajyoti Goswami
Department of Computer Science
Concordia University
Montreal, Quebec, Canada
goswami@encs.concordia.ca

Abstract—**Graph partitioning has important applications in multiple areas of computing, including scheduling, social networks, and parallel processing. In recent years, GPUs have proven successful at accelerating several graph algorithms. However, the irregular nature of the real-world graphs poses a problem for GPUs, which favor regularity. In this paper, we discuss the design and implementation of a parallel multilevel graph partitioner for a CPU-GPU system. The partitioner aims to overcome some of the challenges arising due to memory constraints on GPUs and maximizes the utilization of GPU threads through suitable load-balancing schemes. We present a lock-free shared-memory scheme since fine-grained synchronization among thousands of threads imposes too high a performance overhead. The partitioner, implemented in CUDA, outperforms serial Metis and parallel MPI-based ParMetis. It performs similar to the shared-memory CPU-based parallel graph partitioner mt-metis.**

Keywords- graph partitioning, coarsening, un-coarsening, refinement, GPU

I. INTRODUCTION

Many parallel applications with sparse data structures and data-dependency patterns can be represented by a task interaction graph [1]. This graph may be regularly shaped (e.g. a mesh) or irregular (e.g. a sparse graph).

Formally, a task interaction graph is represented by a tuple (V, E, W_V, W_E), where V the set of vertices (tasks), E is the set of edges, where each edge represents a data interaction link between two incident tasks, W_{v_i} is the computation cost of task v_i, and W_{e_i} is the communication cost among the two incident vertices on e_i. The goal of a graph partitioning algorithm is to divide the graph into k partitions in such a way that each partition is computationally balanced and the total communication costs (edge cuts) among the partitions is minimized [2].

The graph partitioning problem is NP-complete [3]. Consequently, many heuristics have been proposed to optimize the solution [2, 4, 5]. The most successful heuristic for partitioning large graphs used in scientific computations is the multilevel graph partitioning approach [6]. The idea is to first reduce the graph size by matching and collapsing the vertices in multiple coarsening levels until the number of vertices is below a threshold, then the coarsened graph is partitioned, and finally the partitioning is projected back through the multiple levels onto the original graph.

There are several implementations of the parallel graph partitioning algorithm for distributed systems [7, 8, 9, 10, 11]. In these implementations, the graph vertices are divided among the processors and all the coarsening, partitioning, and un-coarsening phases are executed in parallel. During each phase of the algorithm, the processors collaborate on partitioning the graph through message passing.

Due to the serial nature of the coarsening and un-coarsening phases, the quality of the partitions created by this parallel approach may be lower than that of the serial algorithm. However, the parallel schemes deliver significant speedups in comparison to the serial version.

Parallel graph partitioning algorithms have been developed for multi-core architectures [12, 13] as well. Although the shared memory reduces the overhead of inter-process communication in these systems, new challenges such as race-conditions and data coherency arise.

In recent years, GPUs have become widely used for accelerating many codes, including graph processing problems, because of their high computational power, good energy efficiency, and low cost. On programs with regular memory-access patterns and control flow, the speedup delivered by a GPU can be substantial. However, when processing irregular and complex data structures like graphs and trees, parallelization becomes more challenging.

In this paper, we discuss the design and implementation of a multilevel graph partitioner for a CPU-GPU architecture (GP-metis). The partitioner uses the GPU to speed up the intensive parts of its computation. Some GPU applications require graph partitioning to balance the workload among the threads and to increase the parallelism. Using a contemporary partitioning algorithm would require the entire graph to be transferred to the CPU, partitioned there, and moved back to the GPU. Our partitioner eliminates most of this data transfer, thus reducing the transfer latency and improving the performance.

Implementing the graph partitioner on a CPU-GPU system faces several difficulties:

- The GPU threads communicate through global memory. Hence, to keep the memory consistent, we need to synchronize concurrent writes to the global memory. Using atomics or locks for synchronization imposes high overheads and degrades performance. This overhead is much more pronounced on GPUs than on multicore systems because a GPU executes tens of thousands of concurrent threads as compared to a multicore CPU, which only executes tens of concurrent threads.

- The irregularity of the graph structure degrades the performance of the graph-partitioning code running on the GPU. This is because the load distribution among the threads can become imbalanced, which is particularly harmful to performance on GPUs due to their SIMD architecture.
- Memory constraints (GPUs tend to have less memory than CPUs) and the transfer latency between the CPU and GPU make the implementation more challenging.
- Heterogeneous systems combine a SIMD and a MIMD architectural model. Thus, the partitioner need to be decomposed properly to fully exploit the CPU's and the GPU's architecture at the same time.

Our lock-free partitioner focuses on handling the aforementioned challenges through several changes to the existing algorithms, considering the heterogeneity of the architecture, and exploiting the special characteristics of GPUs. To the best of our knowledge, this is the first proposed multilevel graph partitioner designed for and implemented on a heterogeneous CPU-GPU system.

The rest of this paper is organized as follows: Section II explains the contemporary serial and parallel multilevel graph partitioning algorithms. Section III presents our contributions in the design and implementation of a multilevel graph partitioner for heterogeneous architectures. Section VI evaluates and discusses the experimental results. Section V concludes the paper with a discussion of ongoing and future research directions.

II. BACKGROUND

Multilevel techniques [2, 6, 14, 15, 11] for graph partitioning show great improvements in the quality of partitions and partitioning speed as compared to other techniques [4, 5]. Multilevel algorithms are based on reducing the graph size by collapsing the vertices and edges in multiple coarsening phases, subsequently partitioning the coarse graph, and finally un-coarsening it in multiple steps to construct the original partitioned graph.

A. Serial Algorithms

Metis [2], Scotch [15] and Jostle [11] are the most well-known tools for multilevel partitioning. Serial multilevel graph partitioning algorithms consist of three phases: coarsening, initial partitioning, and un-coarsening.

1. Coarsening

Coarsening consists of two steps: matching and contraction. In the first step, a maximal match [2] is calculated. A matching M is a set of edges such that no pair of them is incident on the same vertex. A matching is maximal if it is not possible to add another independent edge to it. There are different methods for calculating a maximal match in the graph, e.g. random matching [14], heavy edge matching [9, 11, 15], and light edge matching [2].

Heavy edge matching (HEM) exhibits the best results where each node is searched for the neighbor connected with the maximum weight edge. The rationale behind this policy is to minimize the weight of the edges in the coarser graph. HEM is the main matching method employed in all serial partitioning algorithms mentioned before (Metis, Scotch and Jostle).

In the contraction step, all the vertex pairs are collapsed together. For two collapsed vertices u and v, the weight of the newly created vertex c (W_c) in the coarser graph is equal to $(W_u + W_v)$. Moreover, if there is one vertex w that is connected to both u and v in the finer graph, then there will be one edge in the coarser graph from w to c with the weight $W_{(u,w)} + W_{(v,w)}$.

The matching and contraction steps terminate based on a specific criterion. In Metis [2] and Scotch [15], the matching ends when the number of vertices of the coarse graph is $O(p)$, where p is the number of partitions, or if the difference in the number of vertices in the coarser graph G_{i+1} compared to the number of vertices in the finer graph G_i is less than a threshold value. Jostle [11] terminates the matching when the number of vertices in the coarse graph is equal to the number of required partitions.

2. Initial Partitioning

Metis [2] applies a Greedy Graph Growing Partitioning (GGGP) algorithm to partition the coarse graph into p parts. It starts from a random vertex and gradually grows a region around it in a breadth-first fashion. Among the possible candidates in every step, the vertex with the largest decrease in the edge cut is chosen first for inclusion in the region. The region continues to grow until it includes almost half of the vertices. By repeating this recursive bisection method, the required number of partitions is obtained.

Scotch [15] employs a similar recursive bisection algorithm. In contrast, the initial partitioning phase of Jostle [11] is trivial as it reduces the number of vertices in the coarsening phase to the number of required partitions.

3. Un-coarsening

The un-coarsening phase has two sub-steps:
- Projection: In this step, the coarser graph is projected back to the finer graph by transferring the partition assignment of each vertex to the corresponding vertices in the finer graph.
- Refinement: A refinement step is performed after each projection to improve the edge-cut using the higher degree of freedom in the finer graph.

Metis [2] and Scotch [15] utilize a modified version [17] of the Kernighan-Lin heuristic [18] for refinement, in which the boundary nodes are sorted based on their gains [2] and are moved between adjacent partitions if doing so reduces the edge cut. The gain of a vertex is defined as the reduction in the edge cut if it moves from partition p_i to partition p_j. However, the balance among the partitions should be maintained after this movement, i.e. no partition should be much larger than any other partition. The projection and refinement steps are terminated when the partitioned original graph is built.

Jostle [11] uses a combined balancing [19] and refinement algorithm. In this approach, a vertex movement from one partition to another is accepted even if it makes the partitions unbalanced. In the following refinement step, the vertex movement is rejected or accepted.

B. *Multilevel Graph Partitioners on Distributed Systems*

Several parallel multilevel graph partitioning algorithms for distributed-memory systems have been proposed [8, 9, 10, 15, 16]. Parallelizing the coarsening and un-coarsening phases is challenging because of their highly serial nature.

ParMetis [10] implements a coarse-grained parallel graph partitioning, which improves performance compared to the fine-grained parallel algorithm [9]. Initially, each processor receives n/p vertices, where n is the number of graph vertices and p is the number of processors in the cluster. The matching phase consists of two passes: in the even numbered passes, each vertex v on processor p sends a match request to its corresponding vertex u on other processors using HEM, but only if $v > u$. Correspondingly, in the odd numbered passes, a vertex v sends its request only if $v < u$. After a few passes, a maximal set is reached and the matching phase terminates. At the end of each iteration, a synchronization step is required in which each processor sends its match requests in one single message to the corresponding processors and subsequently receives the requests from other processors. Based on the received information, the processors decide in parallel how to collapse the vertices to create the next coarser graph.

The initial partitioning phase in ParMetis [10] starts with an all-to-all broadcast of vertices among the processors. Each processor performs a recursive bisection algorithm [2], where the processor completes one branch of the bisection tree. At the end, each processor stores the vertices that belong to its assigned part of the k partitions.

In the un-coarsening step, each processor first projects back its assigned vertices onto the finer graph. Then the same ordering method as in the coarsening step is applied in multiple passes. At the end of each pass, the requests for movement of vertices across the partitions are communicated among the processors, and the movements that do not violate the balance constraints are committed.

PT-Scotch [7, 8] follows a Monte-Carlo approach in the matching phase. Each node sends its match request based on the HEM method with the probability of 0.5. The results show that, after a few iterations, a large part of the vertices are matched. To reduce the communication overhead among the processors, a folding technique is used after several coarsening levels in which the vertices of the coarser graph are duplicated and redistributed to two groups, each to $p/2$ of the processors. The two groups can continue the matching phase independently. This folding process continues recursively $(p/4, p/8, \ldots)$ until each sub-graph is reduced to a single processor. Then a serial recursive bi-sectioning is performed on each processor and the best initial partitioning is chosen for the un-coarsening phase.

During the refinement phase of PT-Scotch, a banded diffusion technique [8, 20] is utilized in which the refinement phase executes on a banded graph extracted from the initial partitioned graph. This banded graph consists of the set of vertices that are located at a specific threshold distance from the partition separators.

Parallel Jostle [16] could face high communication overhead if it continued matching until the number of vertices equals the number of required partitions. So, after reaching a threshold in the coarsening phase, an all-to-all broadcast is executed before each processor continues independently to coarsen down to a single vertex. During the un-coarsening phases, each partition creates its own set of boundary vertices with the same target partition preference, e.g. partition $p_1$constructs a set of its boundary vertices with the preferred target partition p_2. At the same time, partition p_2 creates a similar set of vertices for partition p_1. Consequently, these two sets form an interface region. A serial optimization technique, e.g. KL [18], is executed independently on the different regions. This technique mitigates the communication-intensive vertex movements by isolating different regions of the graph.

C. *Multilevel Graph Partitioners on Shared-Memory Systems*

Gmetis [13] extended a version of Metis to a multicore platform using the Galois programming model [21], which is a sequential object-oriented programming model that supports parallel set iterators. Each Galois iterator may add new elements to the set. However, this approach is found to be not as efficient as ParMetis in terms of performance.

Mt-metis [12] is a multicore graph partitioner based on the Metis algorithm implemented using OpenMP, which achieves better performance than MPI-based distributed graph partitioners. Primarily, the graph vertices are divided among the threads and each thread finds the matches for n/t vertices assigned to it. There is a shared matching vector M among the threads.

Mt-metis employs a lock-free approach to improve the matching performance. In this approach, the matching step is split into two rounds. In the first round, all the threads read from and write to the matching vector freely without any synchronization. Hence, there is a possibility of conflicts in the matching set. In the second round, each thread explores its vertices and the corresponding vertices are matched again to resolve any conflicts.

In the initial partitioning step, each thread partitions the graph into two bisections. Then the best bisection with the minimum edge-cut is selected and half of the threads work on one of the bisection and half of them partition the other bisection recursively [12].

The refinement is performed in two steps as well, and the moving direction of vertices between the partitions is reversed after each round. Moreover, each thread has an assigned buffer for inserting the vertex movement requests of the other threads. At the end of each round, the threads process their buffer and confirm or undo the movements to meet the balance constraints.

III. A MULTILEVEL GRAPH PARTITIONER FOR CPU-GPU ARCHITECTURES

In this section, we discuss the design and CUDA implementation of our parallel multilevel graph partitioner (GP-metis) for a heterogeneous CPU-GPU environment.

Initially, the graph information is copied to the GPU's global memory. In the coarsening step, the vertices are divided among the threads. Each thread scans its assigned vertices and finds their matched vertices. However, due to the possibility of conflicts, another kernel is launched to resolve any conflicts before the contraction begins. The coarsening continues level-by-level until reaching a threshold, beyond which coarsening is faster on the CPU than on the GPU due to the lack of sufficient parallel tasks. Thus, at the threshold level, the coarse graph is transferred to the CPU and the remaining iterations of the coarsening phase are performed on the CPU.

Since the coarse graph size is by definition small, the initial partitioning of the graph has a low level of parallelism. Hence, the initial partitioning phase is also completed on the CPU along with the initial refinement steps until again a threshold level is reached. At this point, the partitioned graph is transferred back to the GPU. The remaining iterations of the un-coarsening phase are executed on the GPU combined with some refinement techniques to recreate the original graph. Fig. 1 shows the proposed graph partitioning scheme for CPU-GPU architectures.

Currently, we assume that the graph size is small enough to fit into the GPU's memory. However, partitioning of bigger graphs that do not fit to the global memory can be done on a cluster of GPUs. This approach will be explored in future work.

To minimize the memory overhead for the graph representation in the GPU's global memory, we use the Compressed Sparse Row (CSR) format [13] to store the graph data structure. The CSR format consists of two arrays: an *adjacency array (adjncy)* of length $2*|E|$ stores the adjacency list of the graph vertices and an *adjacency pointer array (adjp)* of length $|V|+1$, which contains the indices of the adjacency set of each vertex in the adjacency array. Two other arrays, *adjacency weight (adjwgt)* of length $2*|E|$ and *vertex weight (vwgt)* of length $|V|$ are used as well, which contain the weights of the edges and vertices, respectively.

The different phases of the graph partitioning are discussed in detail in the following subsections.

A. *Coarsening*

During the coarsening phase, the vertices of the graph are collapsed to construct the next coarser level of the graph. The coarsening phase consists of two steps: matching and contraction. At each level of coarsening, two arrays are allocated in the global memory on the GPU: a matching array (M) of length $|V|$ that includes the matched pairs to be collapsed in the finer graph (G_i) and a mapping array ($CMap$) of length $|V|$ that stores the vertices' labels in the coarser graph (G_{i+1}).

At the beginning of the matching step, the graph vertices are divided among the threads on the GPU. We are mindful of memory coalescing [22] when distributing the vertices to the threads to improve the memory accesses efficiency. A coalesced memory access is the combination of multiple memory accesses into a single transaction. In modern CUDA-capable GPUs, sets of 32 contiguous threads constitute a warp. When all threads in a warp execute a load instruction, the hardware checks which memory locations the threads access.

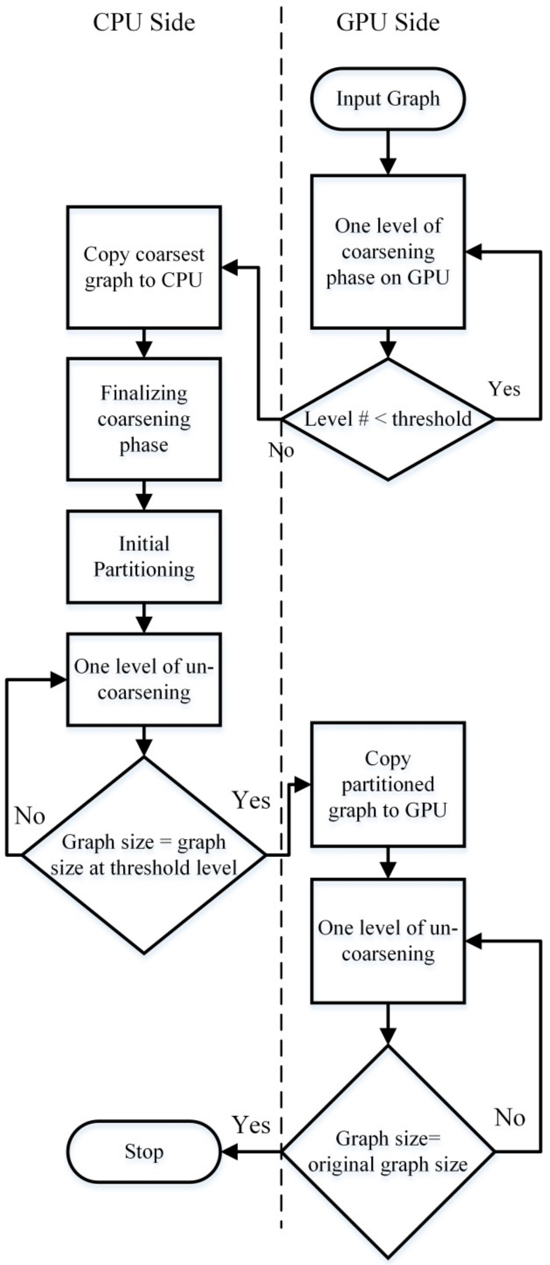

CPU Side
GPU Side
Input Graph
One level of coarsening phase on GPU
Level # < threshold
Yes
No
Copy coarsest graph to CPU
Finalizing coarsening phase
Initial Partitioning
One level of un-coarsening
Graph size = graph size at threshold level
Yes
No
Copy partitioned graph to GPU
One level of un-coarsening
Graph size= original graph size
No
Yes
Stop

Figure 1. Proposed heterogeneous graph partitioning scheme

If all the threads in a warp access locations within a 128-byte block in the global memory, the hardware coalesces the accesses into one transaction. Otherwise, multiple memory transactions have to be issued, resulting in reduced throughput.

Fig. 2 illustrates the memory coalescing technique. If thread 0 accesses vertex n, thread 1 accesses vertex $n+1$ and Thread t_{k-1} (the last thread) accesses vertex $n+(k-1)$, then all memory requests issued by a warp fall into the same 128-byte block. Therefore, they are coalesced, which improves the memory bandwidth significantly.

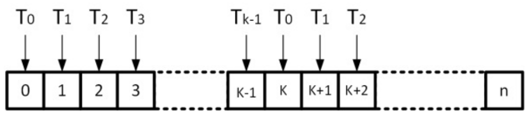


Figure 2. Memory coalescing

To maximize the parallelism in the matching step, we use a lock-free approach similar to mt-metis [12] since fine-grained synchronization incurs too much overhead due to the high number of threads running on a GPU. At the beginning of the matching step, each thread goes over its assigned vertices in the graph and finds their matches using the heaviest edge matching technique (HEM) [2]. If all the edges have the same weight, a random matching (RM) [14] method is used in an iterative fashion where each vertex chooses one of its unmatched vertices randomly to be collapsed in the coarser graph.

All the threads write to the shared matching array (M) concurrently in a lock-free fashion. Hence, there is a possibility that vertex a assumes it has been matched with vertex b while vertex b finds vertex c as its match. Therefore, we need to launch another kernel to resolve these conflicts. In this kernel, each thread goes over its assigned vertices again and checks the match values; if it finds any cases in which $match(a) = b$, but $match(b) != a$, it matches vertex a to itself, $i.e. (match(a) = a)$. This means that vertex a has another chance to find a match in the following coarsening levels. Fig. 3 illustrates the matching step for a graph with eight vertices.

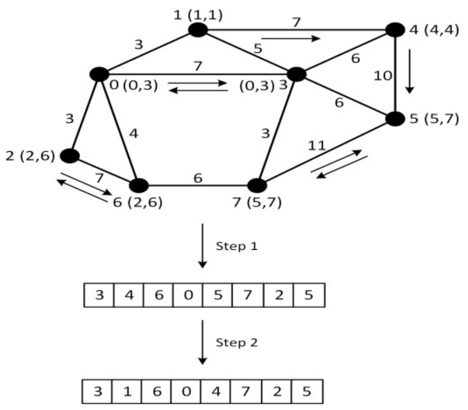


Figure 3. Matching step

Although such conflicts impose some overhead in each iteration of the matching phase due to an increase in the required number of matching iterations, the overhead is significantly lower than using fine-grained locks to resolve conflicts.

The matching array facilitates the creation of the mapping array $Cmap$, which contains the collapsed vertices' labels in the coarser graph. $Cmap$ is constructed by launching the following four kernels on the GPU:

1. Creating the initial $Cmap$: Initially, the $Cmap$ array of length $|V|$ is allocated in the GPU's global memory. Then, a kernel is launched in which each thread executes the following function for all the vertices assigned to it.

```
foreach (vertex i: assigned vertices to this thread)
  if ($i<=M[i]$)
    $Cmap[i]$=1
  else
    $Cmap[i]$=0
```

In this kernel, the $Cmap$ entries are initialized to 0 or 1 depending on the vertices' labels of the matched vertices in the finer graph G_i.

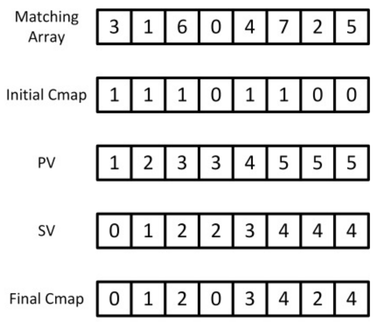
2. Creating a helper array: An inclusive prefix sum is computed on the $Cmap$ array to create the helper array PV. To maximize the performance, we use the parallel inclusive-scan from the CUB library [22], which currently is the highest-performing parallel implementation of prefix sums on GPUs. The last element in the PV array indicates the number of vertices in the coarser graph, $Cgraph$.

3. Subtraction: In this kernel, all the threads subtract one from every entry of the PV array assigned to it in parallel.

4. Creating the final $Cmap$: The final $Cmap$ array is created through the following kernel:

```
foreach (vertex i: assigned vertices to this thread)
  if ($i>M[i]$)
    $Cmap[i]$=$Cmap[M[i]]$
```

It should be noted that the steps needed to create the final $Cmap$ array are computed in-place and we do not need any auxiliary memory space. In addition, all steps are fully parallelized. Fig. 4 illustrates the four steps for creating the final $Cmap$ array. In this example, the number of vertices in $Cgraph$ is 5.



Matching Array 3 1 6 0 4 7 2 5
Initial Cmap 1 1 1 0 1 1 0 0
PV 1 2 3 3 4 5 5 5
SV 0 1 2 2 3 4 4 4
Final Cmap 0 1 2 0 3 4 2 4

Figure 4. Cmap creation steps

In the contraction step, with the help of the $Cmap$ array, the matched vertices are collapsed to form the coarser graph. This step is more complex because each thread calculates a part of the adjacency array ($cadjacency$) and the adjacency weight array ($cadjwgt$) of the coarser graph. To implement the contraction in parallel, each thread should know the start and end indices of its section in the $cadjacency$ and $cadjwgt$ arrays. Thus, these indices need to be calculated beforehand and passed to the threads at the beginning of the contraction step. To accomplish this, two auxiliary arrays ($temp$ and $temp2$) are allocated on the GPU, each of length equal to the total number of threads (T_n), to hold the indices.

In the first launched kernel of the contraction step, each thread calculates the maximum number of entries that it needs in the $cadjacency$ and $cadjwgt$ arrays by scanning all vertices assigned to it. For each vertex v, which is matched with vertex u, the maximum required entries in the coarser graph's adjacency array is the summation of the number of vertices in the adjacency set of v and u. In other words, it is equal to the total number of neighbors of u and v. If the vertex v is matched to itself, then the required number of entries will be equal to the size of its adjacency set. Each thread applies the same logic to each of its assigned vertices and inserts the final number of required entries in the corresponding entry of $temp$.

Next, an exclusive parallel prefix sum is calculated on *temp* to compute the initial start index of each thread in the coarser graph's adjacency list and adjacency weight arrays.

The summation of the value of the last entry of the $temp$ array before and after the prefix sum determines the length of the two required temporary adjacency lists and temporary adjacency weight arrays, $tadjncy$ and $tadjwgt$, respectively, which are allocated on the GPU and are needed for the following reasons. For each two collapsed vertices, some of their neighbors are also collapsed together and the required number of entries for the adjacency list of the newly created vertex in the coarser graph decreases. The temporary arrays $tadjncy$ and $tadjwgt$ facilitate the parallel execution of the threads working on different sets of vertices and help in creating the final $cadjacency$ and $cadjwgt$ arrays.

In the next kernel, each thread starts contracting its vertices and copies the entries of the merged adjacency list and adjacency weight arrays on the $tasjncy$ and $tadjwgt$ arrays beginning from its $index$ in the $temp$ array.

To merge the adjacency sub-lists of vertex i and its match ($M[i]$), we applied two different approaches. In the first approach, the neighbor lists of the pair vertices are merged and sorted using quicksort followed by a *remove* function, which deletes the repeated vertices.

In the second approach, we use a hash table for each thread. Then a hash function is applied to all neighbors of each pair of vertices, which maps the neighbors of two collapsing vertices to the entries in the hash table and constructs the adjacency list of the newly created vertex in the coarser graph.

The hash function reduces the memory space required for the hash table, which ideally should be equal to the number of vertices in the coarser graph. However, to avoid collisions, chaining [23] is used where each bucket of the hash table stores multiple elements, i.e. a clustered hash table.

The hash table approach is faster than the sorting, but it is applicable only when the graph is sparse so that the hash table is not too large to fit inside the GPU memory.

At the end of the contraction, each thread counts the actual number of entries it used for calculating the $cadjcny$ and $cadjwgt$ arrays, and copies the final value to $temp2[tid]$. Then another prefix sum is performed on $temp2$ to calculate the start index of each thread in the final $cadjncy$ and $cadjwgt$ arrays.

Finally, each thread copies the calculated elements from the $tcadjncylist$ and $tcadjwgt$ arrays to the $cadjncy$ and $cadjwgt$ arrays with the help of the $temp$ and $temp2$ index arrays.

At the end of the contraction step, we can free the $tcadjncy$, $tcadjwgt$, $temp$, and $temp2$ arrays. So there is no extra memory overhead for the contraction.

Before going to the next coarsening level, the addresses of all arrays corresponding to the coarser graph are stored in a set of pointer arrays (of length equal to the coarsening level) since they will be needed to project back to the original graph in the un-coarsening phase. Moreover, we reduce the number of launched threads in the following levels of coarsening as the graph size gets smaller. This strategy prevents underutilization of GPU threads due to an insufficient amount of concurrent computations.

The coarsening phase is repeated as it reduces the size of the original graph at each level until reaching a threshold. This threshold specifies the last level in which the coarsening of the graph executes faster on the GPU than the CPU. At this point, the coarse graph is transferred to the CPU and the remaining coarsening steps are completed on the CPU using mt-metis [12]. It should be noted that the size of the coarse graph on the GPU is much smaller than the original graph, making the transfer of the course graph to the CPU very quick.

B. Initial Partitioning

Although the initial partitioning does not need a high level of parallelism, to maximize the performance we use mt-metis to parallelize this phase on a multi-core CPU. Mt-metis has been shown to be faster than other parallel partitioners [12].

C. Un-coarsening

The un-coarsening phase starts on the CPU and continues until it reaches a threshold level. Then the graph information is copied to the GPU and the remaining steps of the un-coarsening phase are executed on the GPU. Mt-metis is used for the un-coarsening on the CPU.

Un-coarsening consists of two steps: Projection and refinement. During the projection step, the coarser graph at level i+1 is projected back to the finer graph at level i. This step can easily be parallelized on the GPU by dividing the vertices of the finer graph among the threads and having each thread specify the partition labels of the projected vertices in the finer graph by considering the *CM* array and saved pointer arrays from the coarsening phase.

The refinement step attempts to improve the graph edge cut by moving the boundary vertices between partitions. This step is more challenging because the concurrent movement of vertices among the partitions may increase the edge cut. With thousands of threads working concurrently, applying lock-based methods for moving vertices between partitions would impose a high synchronization cost and degrade the performance. To overcome this challenge, we use a lock-free approach for refinement.

In the first refinement kernel, the vertices in the finer graph are distributed among the threads and each thread determines the boundary vertices among its assigned vertices. Then it finds the best destination partition for migration of each boundary vertex, if possible. A destination partition is selected for moving a vertex if this move results in the maximum reduction of the edge cut and does not underweight the source or overweight the destination partition. In addition, an ordering method [13] is used that divides each refinement step into two iterations. During each step, vertices can move between the partitions only in one direction (decreasing or increasing). This prevents concurrent exchanges of two vertices between two neighbor partitions, which may result in increasing the edge cut.

To process the threads' concurrent requests for migrating vertices, we allocate a buffer to each partition where the threads insert their movement requests. A request contains the source partition's vertex labels and potential gain. Each buffer has a counter S that indicates the current size of the buffer. To prevent race conditions among the threads, when one thread wants to put a request on a specific buffer, it atomically increments the counter S by one. Thus, multiple threads are able to write to exclusive slots of the buffer concurrently without resorting to locks.

In the next step, an explore kernel is launched with a number of threads equal to the number of partitions where each thread processes the incoming requests in the buffer of its assigned partition. It initially sorts the relocation requests based on their gain. Then it accepts the moves that do not overweight the partition's (say p_i) weight, e.g. for each vertex j, $Weight\ (p_i) + weight\ (v_j)$ will be less than the maximum allowed weight for this partition. The refinement terminates when all boundary vertices are explored.

The refinement at each level repeats for a specified number of passes to improve the edge-cut while keeping the partitions balanced. However, it can be terminated earlier if no move is committed in the current pass. Although the concurrent updates of a partition may unbalance it, the balance of partitions is guarantee by continuing the refinement at the finer graph levels.

D. *Comparison of GP-metis with mt-metis*

It should be noted that GP-metis requires a higher amount of redundant information in the coarsening and refinement phases than mt-metis, i.e. in some of the launched kernels, we need to record more information. The reason is that mt-metis employs a persistent thread paradigm, where data ownership is given to the threads at the beginning of the program and stays the same until the end of the execution.

In contrast, the data ownership in GP-metis is not persistent throughout the execution, in particular, the kernels are launched with a variable number of threads. The rationale behind this is to balance the load among the threads as much as possible and to maximize the performance. Moreover, our GPU implementation uses lock-free approaches to handle conflicts.

IV. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of GP-metis on a CPU-GPU architecture implemented using CUDA/C. We compare the performance of this partitioner with Metis 5.1.0 (a serial partitioner), ParMetis 4.0.3 (a parallel distributed memory partitioner) and mt-metis 0.4.4 (a parallel shared memory partitioner).

We ran all implementations on a system equipped with an Intel Xeon E5540 processor with 8 cores and one Nvidia GeForce GTX Titan GPU, which has 6 GB of GDDR5 RAM. Experiments were performed using four different graphs arising in various areas of computation, which were obtained from DIMACS9 [24] and DIMACS10 [25]. Table I shows the size and specification of these graphs.

For all implementations, we partitioned the input graph into 64 partitions and the imbalance tolerance for each partition was set to 3% (as in Metis [12]).

TABLE I. Input graphs used in the graph partitioner evaluation

Graph Name	*Number of Vertices*	*Number of Edges*	*Description*
ldoor	952,203	22,785,136	Sparse matrix from University of Florida collection
Delaunay	1,048,576	3,145,686	Delaunay triangulation of random points
Hugebubble	21,198,119	31,790,179	2D dynamic simulation
USA Roads	23,947,347	28,947,347	Road network

Fig. 5 shows the speedup achieved by the parallel multilevel graph partitioners on the four graphs. The speedup is the runtime of the parallel graph partitioners relative to the runtime of serial Metis. In each case, we use the minimum runtime of three experiments to compute the speedup.

As Fig. 5 illustrates, GP-metis outperforms Metis and ParMetis on all tested inputs, and its performance is also quite reasonable in comparison to mt-metis (i.e. somewhat better on the larger graphs and somewhat worse on the smaller graphs). The irregularity of the input graph greatly affects the performance of GP-metis, since it increases the workload imbalance between the GPU threads on some of the GPU kernels, which hurts performance.

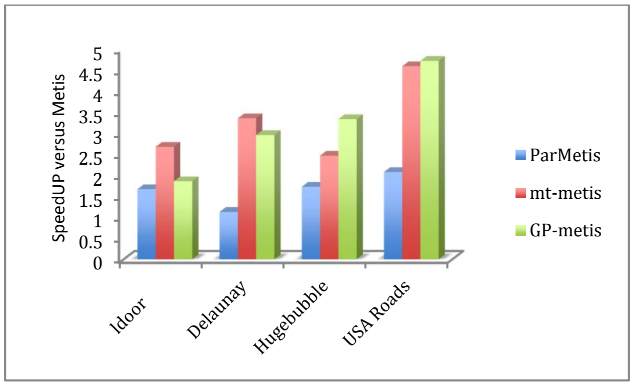


Figure 5. Speedup of ParMetis, mt-metis, and GP-metis over Metis

Table II shows the absolute runtimes of the three parallel graph partitioners. For GP-metis, this time includes the time to transfer the graph between CPU and the GPU. However, I/O times on the CPU are excluded from all timing measurements.

TABLE II. Runtime (in seconds)

Graph	***ParMetis***	***mt-metis***	***GP-Metis***
ldoor	0.77	0.48	0.69
Delaunay	0.65	0.22	0.25
Hugebubble	8.4	5.89	4.36
USA Roads	11.24	5.09	4.95

To validate the comparison of our partitioner with the other parallel partitioners, we also evaluate the ratio of the edge cut achieved by each parallel partitioner relative to Metis. Table III shows the edge cut scaled relative to Metis for the three partitioners.

TABLE III: Edge-cut ratio in comparison to Metis

Graph	***ParMetis***	***mt-metis***	***GP-Metis***
ldoor	1.059	1.057	1.068
Delaunay	1.033	1.027	1.059
Hugebubble	1.134	1.103	1.108
USA Roads	1.177	1.121	1.120

The results show that GP-metis is able to produce partitions of comparable quality to mt-metis and ParMetis. The quality degradation for some of the graphs is due to the finer-grain implementation of GP-metis. In the coarsening and un-coarsening phases of GP-metis, thousands of threads are working concurrently, making the conflict rate much higher in comparison to mt-metis, which only runs a few threads (8 threads in our experiments).

IV. CONCLUSION AND FUTURE WORKS

In this paper, we describe and evaluate a lock-free multilevel graph partitioner for heterogeneous CPU-GPU systems. Some of the challenges we had to overcome in the GPU code are: (1) memory constraints to hold large graphs; (2) the irregular nature of the graph data structure that can degrade the GPU performance; (3) synchronization costs, which are much more pronounced on GPUs running tens of thousands of threads as compared to multi-core CPUs that only run tens of threads; (4) a suitable workload distribution strategy between the CPU and the GPU; (5) a suitable functionality distribution between the CPU and GPU; (6) data-transfer latencies between the CPU and the GPU; and (7) differences in the architectural models between CPUs and GPUs (e.g. MIMD versus SIMD).

To the best of our knowledge, this is the first graph partitioner for hybrid CPU-GPU environments that takes full advantage of the processing power of the GPU cores in both the coarsening and the un-coarsening phases. The experimental results show that our implementation outperforms both Metis and ParMetis and is comparable in performance and quality of the partitions with mt-metis.

In future work, additional optimization strategies will be investigated for better load balancing, since highly irregular inputs can make some kernel loads unbalanced, resulting in performance degradation. Moreover, the partitioning algorithm should be extended to multiple GPUs for handling even larger graphs.

ACKNOWLEDGMENT

This research is supported by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC) and ENCS Faculty Research Support. We also gratefully acknowledge the support of NVIDIA Corporation with the donation of the GeForce Titan GPU used for this research.

REFERENCES

[1] A. Grama, A. Gupta, and G. Karypis, "Introduction to parallel computing: design and analysis of algorithms," *Redwood City, CA: Benjamin/Cummings Publishing Company*, 1994.

[2] G. Karypis, and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing, 20(1)*, pp. 359-392, 1998.

[3] O. H. Ibarra, and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *Journal of the ACM (JACM),* vol. 24, pp. 280-289, 1977.

[4] G. L. Miller, S. H. Teng, and S. A. Vavasis, "A unified geometric approach to graph separators," *In Proceeding of the 32nd Annual Symposium on* Foundations *of Computer Science*, pp. 538-547, 1991.

[5] A. Pothen, H. D. Simon, L. Wang, and S. T. Barnard, "Towards a fast implementation of spectral nested dissection," *In Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pp. 42-51, 1992.

[6] A. Buluc, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent Advances in Graph Partitioning," *CoRR*, pp. 1311-3144, 2013.

[7] C. Chevalier, "PT-Scotch: A tool for efficient parallel graph ordaining," *Parallel Computing*, 34(6–8), pp. 318–331, 2008.

[8] J. H. Her, and F. Pellegrini, "Efficient and scalable parallel graph partitioning," *Parallel Computing,* 2010.

[9] G. Karypis, and V. Kumar, "Parallel multilevel series k-way partitioning scheme for irregular graphs," *SIAM Review 41.2*, pp. 278-300, 1999.

[10] G. Karypis, and V. Kumar, "Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm," *In Procedding of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[11] C. Walshaw, and M. Cross, "Parallel Optimization Algorithms for Multilevel Mesh Partitioning," *Parallel Comput*, vol. 26(12). pp.1635–1660, 2000.

[12] D. LaSalle, and G. Karypis, "Multi-threaded graph partitioning," *In Proceedings of the 27th IEEE International Symposium on Parallel & Distributed Processing (IPDPS),* pp. 225-236, 2013.

[13] X. Sui, D. Nguyen, M. Burtscher, and K. Pingali, "Parallel graph partitioning on multicore architectures," *In Languages and Compilers for Parallel Computing*, pp. 246-260, 2011.

[14] B. Hendrickson, and R. Leland, "A Multilevel Algorithm for Partitioning Graphs," *In Proceedings of Supercomputing'95*, *ACM Press*, pp. 28, 1995.

[15] F. Pellegrini, and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," *In High-Performance Computing and Networking*, pp. 493-498, 1996.

[16] C. Walshaw, and M. Cross, "JOSTLE: parallel multilevel graph-partitioning software–an overview," *Mesh partitioning techniques and domain decomposition techniques*, pp.27-58, 2007.

[17] C. M. Fiduccia, and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," *In Proceedings of the 19th Design Automation Conference*, pp. 175–181, 1982.

[18] B. W. Kernighan, and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal*, vol. 49(1), pp. 291-307, 1970.

[19] S. W. Hammond, "Mapping unstructured grid computations to massively parallel computers," *Technical Report*, 1992.

[20] U. Naumann, and O. E. Schenk, "Combinatorial scientific computing," *CRC Press*, 2012.

[21] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. Chew, "Optimistic parallelism requires abstractions," *In Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* 2007.

[22] Nvidia. CUDA. Retrieved November 21, 2014 from *"http://www.nvidia.com/cuda"*.

[23] M. T. Goodrich, R. Tamassia, and M.H Goldwasser, "Data Structures and Algorithms in Java," 6th edition, John Wiley, 2014.

[24] C. Demetrescu, A. Golberg, and D. Johnson, "Challenge benchmarks," 9th DIMACS Implementation Challenge - Shortest Paths, 2006.

[25] DIMACS 10 challenge graph collection –Graph Partitioning and Graph Clustering, Retrieved January 25, 2015 from *http://www.cc.gatech.edu/dimacs10/downloads.shtml*