An Accelerated Procedure for Hypergraph Coarsening on the GPU

Lin Cheng
Department of Engineering
Trinity College
Hartford, Connecticut, USA 06106–3100
Email: lin.cheng@trincoll.edu

Hyunsu Cho and Peter Yoon
Department of Computer Science
Trinity College
Hartford, Connecticut, USA 06106–3100
Email: {hyunsu.cho, peter.yoon}@trincoll.edu

Abstract—One of the obstacles in accelerating sparse graph applications using GPUs is load imbalance, which in certain cases causes threads to stall. We investigate a specific application known as hypergraph coarsening and explore a technique for addressing load imbalance. The hypergraph is a generalization of the graph where one edge may connect more than two nodes. Many problems of interest may be expressed in terms of optimal partitioning of hypergraphs where the edge cut is minimized. The most costly step in hypergraph partitioning is hypergraph coarsening, the process of grouping nodes with similar connectivity patterns into one node to yield a new hypergraph with fewer nodes. Hypergraph coarsening proves to be computationally challenging on GPUs because many hypergraphs exhibit an irregular distribution of connections. To address the resulting load imbalance, we explore a novel task allocation scheme to distribute work more evenly among GPU threads.

I. INTRODUCTION

A hypergraph is a generalization of a graph where it replaces edges in a graph with hyperedges that connect multiple vertices. It provides a key modeling flexibility that enables accurate formulation of a wide range of computing problems, from VLSI design [1] to social networks [2] and image classification [3], [4]. With the popularity of hypergraphs in recent years, there has been a strong interest to achieve accurate yet effective partitioning in the hypergraph context.

A weighted hypergraph $G = (V, E, w)$ consists of a set of nodes $V$, a set of hyperedges $E$, and the weight function $w : E \rightarrow [0, +\infty)$ that assigns a weight to each hyperedge. Let $E = \{e_0, \cdots, e_{m-1}\}$ and $V = \{v_0, \cdots, v_{n-1}\}$, where $m$ and $n$ are the number of hyperedges and nodes, respectively, and each $e_i \in E$ is a subset of nodes in $V$. See an example hypergraph in Figure 1.

A bipartition of the hypergraph $G$ comprises a subset of nodes $W$ and its complement $\overline{W}$. An optimal bipartition minimizes the edge cut defined by

$$EC(W) = \sum_{e \in W \neq \emptyset} w(e).$$  \hspace{1cm} (1)

Here, we impose a balance constraint so that $W$ and $\overline{W}$ are comparable in size, that is, the imbalance coefficient

$$\epsilon(W) = \frac{2 \cdot \max\{|W|, |\overline{W}|\} - |V|}{|V|}$$  \hspace{1cm} (2)

Fig. 1. An example hypergraph with 6 nodes and 3 hyperedges

is made smaller than the prescribed threshold.

The most computationally demanding step in hypergraph partitioning is hypergraph coarsening, the process of grouping nodes with similar connectivity patterns into one node to yield a simpler hypergraph. A typical hypergraph partitioner is known to spend up to 91% of its processing time on this step [5].

To this end, in this paper, we focus on the hypergraph bipartitioning problem and present a multi-level hypergraph coarsening framework amenable to GPUs. Unlike parallel algorithms for CPU, we modify the algorithms significantly based on GPU threads, resulting in a parallel Mondriaan algorithm [5], the employment of a family of compiler primitives, as well as a parallel suitor matching algorithm.

Experiments show that our GPU-based implementation outperforms the sequential procedure for coarsening for large hypergraphs. Once a good bipartition of the simpler hypergraph obtained by coarsening, it is then used to approximate a bipartition for the original hypergraph. Although we only consider the case of bipartition in this paper, the problem is readily generalized to multiple categories via recursive bipartition.

II. HYPERGRAPH PARTITIONING AND COARSENING

The problem of computing an optimal bipartition of a hypergraph is known to be NP-complete [6]. We adopt a class of approximation algorithms called the multi-level paradigm [1], [5]. First, a series of increasingly coarse approximations to the full problem is computed until the approximation becomes so small that computing an optimal bipartition is easy. The nodes are combined into clusters based on their connectivity patterns. Second, the solution to the coarsest approximation is projected to the previous approximation. Finally, the projected solution is refined to increase the approximation quality. The
A. Hypergraph coarsening

The central piece of the multi-level paradigm is hypergraph coarsening. We compute clusters of nodes so that the connectivity patterns are preserved. Nodes that belong to a similar list of hyperedges should belong to the same cluster. A new hypergraph is then formed whose nodes are clusters of nodes in the original hypergraph. The new hypergraph thus has fewer nodes than the original hypergraph.

We define coarsening of \( G = (E, V, w) \) as an order pair \((V', f)\) where \( V' \) is a set of clusters and \( f : V \rightarrow V' \) is a map that maps nodes to clusters (Line 5 of Figure 2). Later in this section, we will show how to compute a good coarsening map \( f \). Once a coarsening map \( f : V \rightarrow V' \) is determined, the corresponding approximation hypergraph \( G' = (E', V', w') \) is computed (Line 6). First, we compute the new set of hyperedges \( E' = \{e'_0, \ldots, e'_{m-1}\} \) by

\[
e'_i = \{v' \in V' : e_i \cap f^{-1}(v') \neq \emptyset\}.
\]

(3)

In other words, a hyperedge \( e'_i \in E' \) contains cluster \( v' \) whenever the corresponding hyperedge \( e_i \in E \) contains some node belonging to that cluster. Second, we define the new weight function \( w' \) by

\[
w'(e'_i) = w(e_i).
\]

(4)

Note that this is well-defined because \(|E'| = |E| = m\).

After a partition \((W', \overline{W})\) is computed for the approximation hypergraph \( G' = (E', V', w') \) (Line 7 of Figure 2), it is propagated to the original hypergraph \( G \) (Line 8). A partition \((W, \overline{W})\) of \( G \) is finally obtained by

\[
W = f^{-1}(W').
\]

(5)

B. Refinement

After a bipartition \((W, \overline{W})\) of \( G \) is obtained, we refine it for quality. The Fiduccia-Mattheyses algorithm [1] is a heuristic algorithm that lowers the edge cut \( \mathcal{E}(W) \) by moving nodes into and out of \( W \).

C. Connection to weighted graph matching

It remains to determine a good coarsening map \( f : V \rightarrow V' \). To minimize the edge cut \( \mathcal{E}(W) \), we form clusters so that nodes in the same cluster belong to a similar list of hyperedges, as nodes are put in the same categories as their clusters.

We first introduce some basic terminology. The node \( w \) is said to be a neighbor of \( v \) whenever there is a hyperedge \( e \) containing \( v \) and \( w \) is defined to be the total weight of all hyperedges that contain both nodes. To compute similarities, we represent \( G = (E, V, w) \) as an \( m \times n \) sparse matrix \( A = [a_{ij}] \) where

\[
a_{ij} = \begin{cases} 1 & \text{if } v_j \in e_i \\ 0 & \text{if } v_j \notin e_i. \end{cases}
\]

Here, we use the fact that \( a_{ij} \) and \( a_{jk} \) are both 1 if and only if both \( v_i \) and \( v_j \) belong to hyperedge \( e_k \). Define the weighted dot product \((\cdot, \cdot)_w\) by

\[
(a_i, a_j)_w = \sum_{k=0}^{m-1} a_{ik}a_{kj}w(e_k).
\]

(8)

where \( a_i \) denotes the \( i \)-th column vector of \( A \). Then the similarity between nodes \( v_i \) and \( v_j \) is simply \((a_i, a_j)_w\).

The problem of computing \( f : V \rightarrow V' \) is now reduced to the weighted matching problem if \( f \) maps at most two nodes to each cluster in \( V' \). Let us define the metric closure \( M(G) = (E_M, V, \zeta) \) of \( G = (E, V, w) \) as follows:

- \( M(G) \) is an ordinary graph: \(|e| = 2\) for each \( e \in E_M \).
- \( v_i \) and \( v_j \) are connected by an edge \( e_{ij} \in E_M \) if and only if they are neighbors.
- If \( e_{ij} \) is the edge connecting nodes \( v_i \) and \( v_j \), then \( \zeta(e_{ij}) = (a_i, a_j)_w \).

An optimal weighted matching of \( M(G) \) matches each node in \( V \) to an adjacent node, so that the total weight of the matched edges is maximized. Since we wish to cluster nodes with a high level of similarities between them, we set \( f(v_i) = f(v_j) = \min\{i, j\} \) whenever \( v_i \) is matched to \( v_j \) under the weighted matching problem. We set \( f(v) = v \) if \( v \) is left unmatched.

III. RELATED WORK

There has been an extensive research on the subject of weighted matching. For example, several parallel algorithms for the weighted matching problem are described in [7], [8]. Both papers assume that the adjacency lists for the nodes are available for constant-time lookups. This assumption holds for ordinary graphs – it takes a constant time to find all neighbors of a given node. However, the assumption fails to hold for hypergraphs. Since adjacency in hypergraphs is defined only implicitly, to find neighbors of \( v \), we must scan all hyperedges containing \( v \) and collect the nodes in those hyperedges. Hence, it takes at least linear time to look up the neighbors. Auer [5] adapted weighted matching algorithms to hypergraph partitioning. A
class of approximate matching algorithms [9], [10] is coupled with a class of neighbor-finding algorithms. The adjacency lists are computed as they are needed and the metric closure is only implicitly accessed. Some of the matching and neighbor-finding algorithms are inherently sequential, however.

Çatalyürek et. al. [11] presented two parallel algorithms for hypergraph coarsening on multi-core CPUs. One uses atomic lock operations to prevent inconsistent matching decisions made by multiple threads. Another allows threads perform matchings as they would in a sequential setting and later resolves incompatible matchings later. Both algorithms achieve a linear speedup with respect to the number of physical CPU cores. The algorithms must be modified significantly for GPUs, however, because they operate under the assumption that threads are not as independent. We shall present a way to overcome this limitation in Section V.

IV. AN ALGORITHM FOR HYPERGRAPH COARSENING

We first describe a sequential implementation of hypergraph coarsening. The implementation consists of two algorithms: the Mondriaan algorithm and the Greedy matching algorithm.

A. Mondriaan algorithm

The Mondriaan algorithm [5], described in Figure 3, computes the list of all neighbors $v_k$ of a given node $v_j$ and the associated similarities $(v_j, v_k)_w$. Rather than computing $(v_j, v_k)_w$ for all columns $k$, we exploit the sparse representation of the matrix $A$. First, we scan the nonzero entries of the $j$-th column $a_j$ to construct a list of all hyperedges that contain $v_j$. Second, for each hyperedge $e_i$ containing $v_j$, we scan all nonzero entries of $e_i$ to construct a list of all nodes $v_k$ in $e_i$. If both $v_j$ and $v_k$ are in $e_i$, then the nodes share one hyperedge and the similarity score $S(k)$ is incremented by $w(e_i)$. We repeat this process for all other hyperedges $e_i$’s, so that each $S(k)$ would eventually contain the total weighted sum of all hyperedges containing both $v_k$ and $v_j$, namely $(v_k, v_j)_w$.

B. Greedy matching algorithm

A greedy approach to matching is outlined in Figure 4. In this algorithm each $v_j$ is matched to the neighbor $v_K$ most similar to it. As soon as $v_j$ and $v_K$ are matched, they are removed from the pool of nodes available for matching.

V. GPU IMPLEMENTATION

A. GPU fundamentals

In recent years, GPUs have gained popularity as a general-purpose parallel accelerator. GPUs adopt a many-core architecture where a large number of hardware threads deliver a high instruction throughput. To simplify control logic, GPUs adopt the single-instruction-multiple-data (SIMD) paradigm where a group of hardware threads executes an identical set of instructions on multiple portions of data.

In particular, hardware threads on NVIDIA GPUs are organized in *warps*, or units of 32 threads. Threads in the same warp share a single instruction counter. If all threads in a warp execute an identical instruction on adjacent data elements, one single vector operation is loaded to the shared instruction counter. On the other hand, if only some threads in a warp execute a given instruction depending a specified condition, the corresponding operation is still loaded to the shared instruction counter but the remaining threads stall. The inactive threads become active once the instruction counter receives a new operation that applies to those threads. Consequently, in cases where a few threads receive a lion’s share of workload, the other threads in the same warp remain stalled much of time. This phenomenon is known as *warp divergence* and has a debilitating effect on performance. A detailed coverage of

---

**Input:** A sparse matrix $A$ induced by $G = (E, V, w)$; the associated weight function $w$; the index $j$ of the node $v_j$ whose neighbors are to be computed; and an array $mate$ whose $k$-th entry gives the index of the node to which $v_k$ is currently matched ($mate(k) = -1$ if unmatched yet).

**Output:** A set $V$ of unmatched neighbors of node $v_j$; and an array $S$ where $S(k) = (a_j, a_k)_w$

**Procedure Mondriaan**($A$, $w$, $j$, mate)

1. $S(k) \leftarrow 0$ for all $0 \leq k < n$
2. for each row index $i$ for which $a_{ij} = 1$
3.   for each column index $k$ for which $a_{ik} = 1$
4.     if mate($k$) = -1 and $k \neq j$ then
5.         $S(k) \leftarrow S(k) + w(e_i)$
6.     end if
7. end for
8. end for
9. return $(V, S)$

**Procedure Compute-Matching**($A$, $w$)

1. if mate($j$) = -1 then
2.     $(V, S) \leftarrow $Mondriaan($A$, $w$, $j$, mate)
3.     $K \leftarrow \max_k S(k)$
4.     if mate($K$) = -1 then
5.         $mate(j) \leftarrow \max_{j,K}$
6.     end if
7. end if
8. return mate
9. end procedure
Input: A sparse matrix \( A \) induced by \( G = (E, V, w) \); the associated weight function \( w \); the index \( j \) of the node \( v_j \) whose neighbors are to be found; a function object \( F \), to be called with the index of each neighbor of \( v_j \).

1: procedure Mondriaan-Parallel \((A, w, j, F)\)  
2:     for each row index \( i \) for which \( a_{ij} = 1 \) in parallel do  
3:         \( \omega \leftarrow 32 \) \( \triangleright \) number of threads per warp  
4:         \( l \leftarrow \) the lane-id of the current thread \((0 \leq l < \omega)\)  
5:         \( a \leftarrow \text{nnz}(e_i) \) \( \triangleright \) \# of nonzero entries  
6:         \( b \leftarrow \text{INCLUSIVE-SUM-SCAN-WARP}(a) \)  
7:         \( S \leftarrow \text{SHFL}(b, \omega - 1) \)  
8:         \( b' \leftarrow b - a \) \( \triangleright \) convert to exclusive scan  
9:         \( L \leftarrow \lfloor S/\omega \rfloor(l + 1) \)
10:     for \( z \) from 0 to \( \omega - 1 \) do  
11:         if \( \text{SHFL}(b', z) < L \) then  
12:             \( E_r \leftarrow z \)
13:         end if  
14:     end for  
15:     \( E_c \leftarrow L - \text{SHFL}(b', E_r) \)  
16:     \( S_r \leftarrow \text{SHFL-UP}(E_c, 1) \)  
17:     \( S_c \leftarrow \text{SHFL-UP}(E_r, 1) \)  
18:     if \( l = 0 \) then  
19:         \( S_r \leftarrow 0 \)  
20:     end if  
21:     \( S_c \leftarrow S_r + \) [thread-id of first thread in this warp]  
22:     \( \text{Now } (S_r, S_c) - (E_r, E_c) \) is the range of indices for nonzero entries assigned to current thread  
23:     for each nonzero \( a_{ik} \) assigned to current thread do  
24:         Call \( F(k, \omega(e_i)) \) \( \triangleright \) pass weight of hyperedge \( e_i \)  
25:     end for  
26: end for  
27: end procedure

Fig. 5. The parallel Mondriaan algorithm with collaborative task planning  

For workload, we see that nonzero rows carry wildly varying amount of work. To eliminate warp divergence that would result otherwise, we employ a collaborative planning to even out workload among the 32 threads in each warp. First, the 32 threads collect all nonzero entries in the 32 nonzero rows assigned to them. The threads then take equal shares of the nonzero entries. Each thread carries a range of 2D indices indicating the set of nonzero entries that it must process. See Figure 5 for details.

For the collaborative planning step, we employ a family of compiler primitives known as Shuffle [13], which is implemented in the current version of the CUDA toolkit [14]. Since NVIDIA GPUs operate under the SIMD paradigm, it is useful to think of local registers as a block of memory cells rather than an isolated box. A single instruction is essentially a vector operation acting on a block of cells (see Figure 6a). The Shuffle primitives let the programmer shuffle the contents of the block (see Figure 6b). In addition to providing inter-warp communication, the Shuffle primitives also act as a lightweight, fine-grained synchronization mechanism — an operation is applied on a group of 32 threads in a synchronous fashion with a single hardware instruction. In addition, common parallel primitives such as reduction and prefix sum are readily implemented using Shuffle [15].

Consider a simple example in Figure 7. In this example, we wish to find neighbors of node 3. (For space consideration, let the warp carry 4 threads instead of the usual 32.) Taking advantage of sparsity, we only consider hyperedges 1, 2, 4, and 6. To distribute workload, we collect all nonzero entries in those hyperedges and divide them into four equal portions. Since hyperedges 1, 2, 4, and 6 contain 12 nonzero entries combined, each thread should get three nonzero entries.

The first step is to compute a fair share of nonzero entries for each thread. We treat local variables as arrays indexed by lane ID. Let \( a(l) \) be the number of nonzero entries in the \( l \)-th lane, and let \( b(l) \) be the number of all nonzero entries combined in the first \( l \) rows (Lines 5-6 of Figure 5). Then \( b(3) \) gives the total number of nonzero entries in the four rows, so set \( S \) to...
be that value (Line 7). Then the fair share is given by $\lceil S/\omega \rceil$. The array $L(\cdot)$ in Line 9 has a special meaning: number all 12 nonzero entries from 0 to 11, and $L(l)$ would be one past the index of the last entry being assigned to thread $l$. For instance, thread 0 would receive entries 0 to 2, thread 1 would receive entries 3 to 5 and so forth.

Once we find the range of entries being assigned to each thread, it only remains to convert the range into a pair of 2D coordinates in the matrix. For example, we would like to locate the row and column containing entry 3. The loop in Lines 10-14 sets $E_r(l)$ to be the largest index for which $b'(E_r(l)) < L(l)$. The effect is that $E_r(l)$ gets the row index of entry $L(l)$. So $E_r(0) = 1$ indicates that entry 3 is in row 1. Now that the row indices for ranges are known, let us find the corresponding column indices. Looking at the exclusive sum scan, we realize that $b'(E_r(l))$ gives the number of nonzeros in the rows preceding row $E_r(l)$, namely rows 0 to $E_r(l) − 1$. Hence, subtracting $b'(E_r(l))$ from $L(l)$ gives the desired column index for entry $L(l)$ (Line 15). For instance, entry 3 is in column 2. Once $(E_r,E_c)$ is found, shift the arrays to the right by one to obtain $(S_r,S_c)$ (Lines 16-21).

C. Parallel suitor-matching algorithm

To divide the matching task into parallel pieces, matching decisions should be made by multiple threads at once. Hence, we cannot rely on node removal to prevent conflicting decisions, as described in Figure 4. The parallel algorithms discussed in Section III adopt the suitor approach, where the nodes make preliminary matching decisions, and only compatible decisions are made final. More specifically, each node $v$ makes a tentative proposal to pair with the node $w$ that is most similar to $v$. If $w$ also makes the counter-proposal to pair with $v$, the proposals are made final, and $v$ and $w$ are matched. We adopt the queue-based suitor-matching algorithm presented in [7]: whenever $v$ is matched with another node, we enqueue $v$ to the queue $Q_C$. After the first round of compatible proposals are processed, the neighbors of nodes in $Q_C$ will be considered for additional matching. See Figure 9 for details.

In the second phase of the algorithm, we make a list of all neighbors for the nodes in $Q_C$. We recompute matching proposals for those neighbors that tried to pair with nodes in $Q_C$ but did not succeed. Since the best candidates have already been matched away, the neighbors will be matched with their second-best candidates. From the matching proposals, collect all compatible ones and make them final. Enqueue to $Q_C$ all newly matched nodes so that their unmatched neighbors would be considered in the following round. Repeat the second phase of the algorithm until there is no more matching to be made, i.e. until $Q_C$ becomes empty.

VII. EXPERIMENTAL RESULTS

We tested our GPU implementation with sparse matrices from [16]. All experiments were done on a dual 2.0 GHz Intel® Xeon® E5-2620 CPU and four NVIDIA® Tesla® K20c GPUs with CUDA 6.5. Each GPU has 2,496 cores divided into 13 streaming multiprocessors and a total memory of 5 GB. We compare the performance of our GPU implementation with the sequential implementation of Mondriaan neighbor-finding algorithm [17]. Other sequential neighbor-finding algorithms offer better performance but lack parallel counterparts.

We observe in Table III that hypergraphs with a long-tailed distribution of nonzero entries exhibit good speedups. One of the major factors is the number of nonzero entries in each column, i.e. the number of hyperedges that contain each node. In Table II, we see that for all four data sets, over 90% of the columns have fewer than 32 nonzero entries, i.e. most nodes belong to fewer than 32 hyperedges. The difference between the first two sets and the last two is the presence of heavily connected nodes. The sets flickr and wikipedia contain so many outliers that the arithmetic mean is far larger than the median. On the other hand, the sets stanford and stanford-berkeley do not contain as many outliers. Unfortunately, our parallel implementation exhibits a good speedup for the first two sets but not for the last two.

To verify our speculation, we generated a synthetic hypergraph with an extreme long-tailed distribution of nonzero entries. Let $A$ be a $700,000 \times 700,000$ sparse matrix with zeros everywhere except for the first 1,000 columns. Each of the 1,000 columns have exactly 100,000 entries of 1 randomly distributed in the column and zeros everywhere else. For this matrix, our GPU implementation computed the coarsening map in 262 seconds, whereas the sequential reference in 32,240 seconds!

TABLE II. DISTRIBUTION OF NONZERO CARDINALITY OF COLUMNS

<table>
<thead>
<tr>
<th>Input</th>
<th>Min</th>
<th>Q1</th>
<th>Median</th>
<th>Q3</th>
<th>Max</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>flickr</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>8549</td>
<td>12</td>
</tr>
<tr>
<td>wikipedia</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>75547</td>
<td>12</td>
</tr>
<tr>
<td>stanford</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>9</td>
<td>255</td>
<td>8</td>
</tr>
<tr>
<td>stanford-berkeley</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>12</td>
<td>249</td>
<td>11</td>
</tr>
</tbody>
</table>

TABLE III. SPEEDUP OVER SEQUENTIAL COARSENING ALGORITHM [17]

<table>
<thead>
<tr>
<th>Input</th>
<th>GPU (s)</th>
<th>Sequential (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>flickr</td>
<td>58.69</td>
<td>876.31</td>
<td>14.93</td>
</tr>
<tr>
<td>wikipedia</td>
<td>50.42</td>
<td>810.60</td>
<td>16.08</td>
</tr>
<tr>
<td>stanford</td>
<td>145.54</td>
<td>65.06</td>
<td>0.45</td>
</tr>
<tr>
<td>stanford-berkeley</td>
<td>625.60</td>
<td>40.79</td>
<td>0.07</td>
</tr>
</tbody>
</table>

VIII. CONNECTION TO IMAGE CLASSIFICATION

Recent work in computer vision [3], [4] casts the problem of unsupervised image classification as that of hypergraph partitioning. The idea is that hyperedges capture higher-order relationships among sample images that pairwise connections do not.

We outline the steps of obtaining a weighted hypergraph from a given set of images to be classified. For simplicity, we assume two categories of images.

1) Extract local features of sample images using transformation-invariant descriptors such as SIFT [18].
2) Pool local features into one summary vector for each image. The Locality-constrained Linear Coding (LLC) algorithm [19] first converts local features into short codes indicating their positions relative to one another. The codes are then pooled and concatenated to produce one final summary vector for each picture. The process is designed such that information relating the dominant object in the picture is well represented while extraneous backgrounds and noise are pruned.
a more general task allocation strategy which will lead to a single column. Assigning multiple columns to GPU threads of earlier versions of software.

like to thank NVIDIA Corporation for providing GPUs under College under Student Research Program. The authors would also like to thank Sam Johnson for his assistance with the development of earlier versions of software.

VIII. CONCLUSION

This paper presented an accelerated procedure for hypergraph coarsening. A novel task planning scheme was proposed to boost performance on NVIDIA GPUs, where instruction counters are shared by multiple hardware threads. Our GPU implementation outperformed a comparable sequential implementation for hypergraphs which contain heavily-connected nodes. However, when nodes are evenly connected, our implementation did not perform as well as expected.

We ascribe the discrepancy to static task allocation: Each column of \( A \) is assigned at least 32 threads. For columns with many nonzero entries, the threads are fully utilized, while for columns with nonzero entries fewer than 32, the threads are under-utilized and thus stall for the lack of work. The problem occurs because each thread is statically assigned a single column. Assigning multiple columns to GPU threads should address this issue. To that end, we are investigating a more general task allocation strategy which will lead to a higher level of thread utilization.

ACKNOWLEDGMENT

The research of Hyunsu Cho was supported by Trinity College under Student Research Program. The authors would like to thank NVIDIA Corporation for providing GPUs under CUDA Teaching Center Program. The authors would also like to thank Sam Johnson for his assistance with the development of earlier versions of software.

```
1: procedure LAMBDA 1(k, inc; j, S, mate, K)
2: l ←[thread-id of current thread]
3: max_score ← 0
4: max_id ← −1
5: if mate(k) = −1 then
6: ATOMICADD(S(k), inc)
7: if S(k) > max_score then
8: max_score ← S(k)
9: max_id ← k
10: end if
11: end if
12: K ← max_id
13: end procedure

14: procedure LAMBDA 2(k, : j, S, Q_t, mate)
15: if k ≠ mate(j) and cand(k) = j
16: Enqueue k to Q_t
17: end if
18: end procedure

Fig. 8. Helper lambdas (templates for function objects)

3) Compute pairwise distance between nodes (input pictures) using the associated summary vectors.
4) For each \( n \)-th sample, collect \( k \) nearest neighbors and put them in the \( n \)-th hyperedge.
5) Assign weights to hypergraphs to reflect the compactness of member nodes [20].
6) Compute a bipartition for the resulting hypergraph.

```

Input: A sparse matrix \( A \) induced by \( G = (E, V, w) \); and the associated weight function \( w \)
Output: An array \( \text{mate} \) whose \( k \)-th entry is the index of the node \( v_k \) being paired with \( \text{mate}(k) = −1 \) if unmatched

1: procedure COMPUTE-MATCHING-PARALLEL(A, w)
2: mate(j) ← −1 for all \( 0 \leq j < n \)
3: cand(j) ← −1 for all \( 0 \leq j < n \)
4: array to store tentative matching decisions
5: Phase 1
6: for \( j \) from 0 to \( n − 1 \) do
7: \( F_1 ← \text{LAMBDA} 1(\cdot, \cdot; j, S, \text{mate}, K) \)
8: \( \text{MONDRIAN-PARALLEL}(A, j, F_1) \)
9: end for
10: Phase 2
11: for \( j \) from 0 to \( n − 1 \) in parallel do
12: if \( \text{cand}(j) = \text{cand}(\text{cand}(j)) \) then
13: \( \text{mate}(j) ← \text{cand}(j) \)
14: Enqueue \( j \) to queue \( Q_C \)
15: end if
16: end for
17: while \( Q_C \) is not empty do
18: \( F_2 ← \text{LAMBDA} 2(\cdot ; \cdot; j, S, Q_t, \text{mate}, \text{cand}) \)
19: \( \text{MONDRIAN-PARALLEL}(A, j, F_2) \)
20: end for
21: for \( j \) in \( Q_t \) do
22: \( F_3 ← \text{LAMBDA} 1(\cdot, \cdot; j, S, \text{mate}, K) \)
23: \( \text{MONDRIAN-PARALLEL}(A, j, F_3) \)
24: end for
25: for \( j \) from 0 to \( n − 1 \) in parallel do
26: if \( \text{cand}(j) = \text{cand}(\text{cand}(j)) \) then
27: \( \text{mate}(j) ← \text{cand}(j) \)
28: Enqueue \( j \) to queue \( Q_N \)
29: end if
30: end for
31: \( Q_C ← Q_N \)
32: \( Q_N ← \text{empty queue} \)
33: \( Q_t ← \text{empty queue} \)
34: end while
35: return \( \text{mate} \)
36: end procedure

Fig. 9. The parallel suitor matching algorithm

978-1-4673-9286-0/15/$31.00 ©2015 IEEE
REFERENCES


[14] CUDA C programming guide. NVIDIA.

[15] The CUB library. NVIDIA.


978-1-4673-9286-0/15/$31.00 ©2015 IEEE