Graph Combing

Final Report

Saichand Bandarupalli
colorado school of mines
bandarupalli@mines.edu

1. Introduction

GPU accelerator for computing the graphs on GPUs. The GPU accelerator main function is to preprocess the Graph in a way to make the Graph suitable for efficient GPU processing. The accelerator goes through the graph and finds the vertices which are having skewed degree vertices. The no of in edges to the vertices is compared with the threshold number of in edges. If the in edges number is less than the threshold the preprocessor moves to the next part of the graph. But, if the number of edges for the vertex is above the threshold the accelerator creates a copy of the vertex for the edges which are above the threshold.

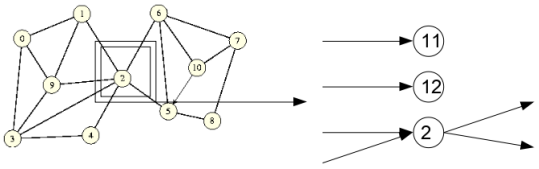

figure 1: The transformation of edge 2 after applying the pre processing

This process of combing the graph is carried out throughout the Graph. The second important feature of the accelerator is the combing function. The combing function can replicate and perform the same operation on all the children vertices of the vertex and at the same time diminish the number of parallel functions in case of graph divergence. The input to the preprocessor is a graph with many high degree vertices. Output of the preprocessor is a graph with less high degree vertices. The pre processor tries to maintain the threshold width which is the number of out edges each vertex of the graph can have.

The explanation above gives a quick overview of how the graph combing algorithm functions. The detailed working of the Graph combing preprocessing algorithm is discussed in the methodology section. The idea of processing graphs by combing them and resultantly reduce the number of high degree vertices is inspired by how GPU will be processing or executing the instructions. Here as we can observe reducing the number of out going edges by graph combing we are able to increase the amount of parallelism in the graph application.

The main motivation behind developing the graph combing algorithm is to improvise on the bottlenecks of GPU computing especially when processing graph applications. The main bottle necks to GPU computing kernel invocations, Performance bottlenecks,SRAM resource utility and etc other bottlenecks which are degrading the performance of Parallel processing on GPUs are discussed in detail in the section2. The methodology section discusses the working model of the graph combing with examples and pseudo codes to explain the nuances of variations of the graph combing methodology

The section 4 covers the comparison results. the Graph combing is performed on the eu-email data set which consists of 1005 nodes and 25000 edges. the connected components problem is taken to compare the performance of the data set with combing and without combing. Further the results section also discusses why the new improvement added to the graph combing enables a increase in the applicability of the graph combing. Finally the future improvement and conclusion is covered in section 5 where the discussion of how this idea can be extended and applied to Graph chi system for enabling speed processing of graphs on GPUs by increasing the usage of memory down the hierarchy and also the constant and shared memory.

2. Background:Bottlenecks to GPU computing

2.1 Kernel invocations:

When we have a GPGPU working model. The number of kernel function calls invoked by the CPU on GPU when executing Graph applications are way higher compared to non graph applications. Graph applications require frequent synchronisations between the subgraph and the graph on the disk. Thus resultantly require CPU interventions to provide synchronisation capability.

For each kernel invocation the amount of computation carried out is far less when compared to non graph applications. In the CPU and GPU systems the communication between CPU and GPU is carried out using the PCI interface. This bears long latencies to the process which is involved in communication between the GPU and the CPU. This is because of the high responsive time of the PCI interface. In addition to this the messages carried by the process

, .

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

are really small. Resultantly, creating large amount of overhead per unit process.

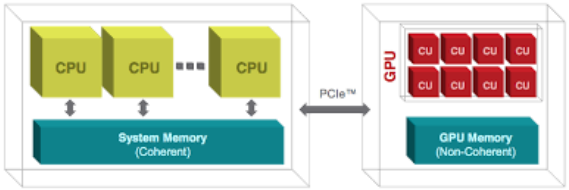

figure 2: The CPU-GPU model with PCI interface

Data transfer in Graph applications are very frequent. Which further worsens the overhead for processing a process unit. When the streaming multiprocessors finished processing on the assigned vertices. The next set of vertices to process is determined only when all the other streaming multiprocessors finished their work assigned in the kernel. because there can be dependencies among the vertices processed in multiple streaming multi processors. As the GPUs does not support the Global synchronisation among the various streaming multiprocessors (SMs), the graph applications use kernel invocations for Global synchronisation.

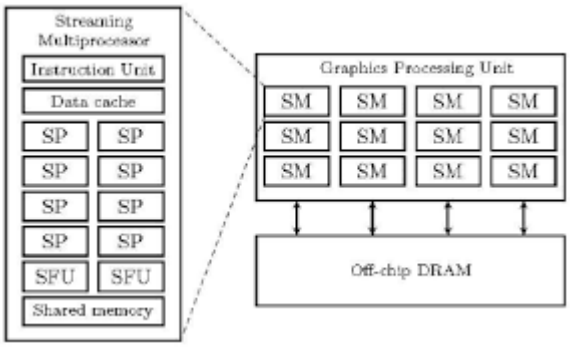

figure 3: The GPU and the streaming multiprocessor assembly

The output data needs to be updated in the GPU memory so that vertices processed in the next kernel can read the data appropriately. Thus with the frequent CPU-GPU interactions total overhead on PCI transfers increases the average processing time of the process in Graph applications.

2.2 Performance bottlenecks

The long memory latency is a major cause for majority of pipeline stalls in graph applications. Graph applications by nature work with large data-sets, and hence they are likely to experience higher memory related stalls. In Addition to the large data the transfer of data from CPU to GPU frequently is another cause of performance bottleneck for GPUs. Further, the data cached from one super-step is unlikely to be useful in the second super-step. This characteristic of graphs almost kills the functionality of L1 and L2 caches down the memory hierarchy in GPUs. In conclusion, Due to the ineffective cache usage, the impact of long memory latency is higher in the graph applications.

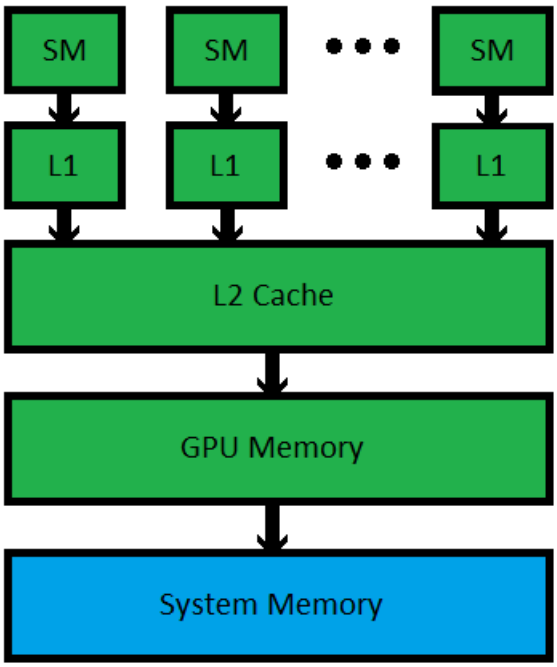

figure 4: The memory hierarchy of GPU

2.3 SRAM resource sensitivity

Across all the graph applications, the register file is the most effectively leveraged SRAM structure among the three. Graph applications tend to use a fairly less amount of shared memory and constant memory. Shared memory is smaller and faster memory that can be shared by all the threads that are running within a CTA, while global memory is the large but slow memory that can be accessed by all SMs. Unlike traditional CPUs, GPU put the burden on applications to explicitly manage the usage of shared and global memory.

When an application wants to use the shared memory then it executes a move instruction to move the data from global memory to shared memory. Now, when the data is available in the shared memory then the application can issue a second load instruction to bring the data into the execution unit. The data that is moved into the shared memory is aimed at providing reuse of data to the execution lanes. If there is not enough reuse of data then the entire overhead involved in moving data from global memory to shared memory and then to the execution lanes actually consumes more time than simply loading data directly from global memory.

Thus in the absence of sufficient data reuse, shared memory access only increases the memory access time as well as the instruction count as the data in the global memory needs to be loaded to the shared memory first by a load instruction and then another load instruction should be executed to get the data from the shared memory to the register. Therefore, in the relatively short kernel functions that are used in graph applications, it is hard to effectively leverage the shared memory Thus graph applications do not try to exploit the shorter latency shared memory and instead simply load data from global memory.

Constant memory is a region of global memory that is cached to the read-only constant cache. Given that GPUs L1 cache size is rel-

atively small, maintaining some repeatedly accessed read-only data in constant cache helps to conserve memory bandwidth. However, constant cache is typically the smallest among the SRAM structures embedded in the GPU die. Therefore, to gain performance benefits from using constant cache, programmers should carefully decide which data to store in the constant memory. Given such an extremely high cache miss rate. This means that L1 cache is entirely ineffective for graph processing. The condition is the same if even the size of L1 cache is increased.

The reason for this ineffectiveness of caches can be inferred from the fact that between two kernel invocations, the CPU has to do memory transfer on GPU memory. Hence, each kernel invocation essentially loses any cache locality that was present at the end of prior kernel invocation.

2.4 SIMT line utilisation

The processing of graphs leads to variable amounts of parallelism during the execution. The Graph comprises of vertices of variety of degrees (i.e. the number of edges). The execution of vertices with such multitude degree edges is to loop over each vertex edges. If each vertex is processed by one SIMT lane so that multiple vertices are processed by a warp in parallel, then the number of iterations executed by each SIMT lane varies as the degree of each vertex varies. This leads to a significant diverged control flow. Thus the SIMT lane utilization varies significantly in graph applications.

2.5 Load imbalance

coarse grain load distribution and the fine grain load distribution are the parameters that can be used to measure the load balancing in the thread blocks and the load balancing within the wrps. Coarse-grain load distribution in many graph application is well balanced once the input data is large enough. However, the fine grained load distribution that is measured across the CTAs, warps, and SIMT lanes exhibits higher levels of imbalance. As briefed earlier, vertices in a graph have multitude number of edges attached. Therefore, the amount of tasks that needs to be processed by each vertex is different and hence the load is imbalanced in graph applications. Given that the CTA execution time is determined by the longest warp execution time and kernel execution time is only determined by the longest CTA execution time, such load imbalance can significantly degrade the overall performance.

3. Methodology:The Graph combing Method and its variations

The Graph combing methodology involves creating new nodes for vertices which have greater number of outgoing edges than the threshold. This process of splitting up the nodes into new nodes is carried throughout each vertex of the graph. The figure below explains the working of the Graph combing algorithm with a example

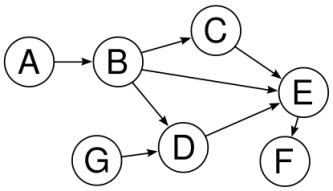

figure 5: The example graph before applying the preprocessing

Vertices with unequal number of in and out edges. Let us suppose we have I in edges and O out edges. The combing operation is carried out accordingly. If the number of in edges are greater than the number of out edges. n number of new vertices are created. Where n is the difference of number of in edges and the threshold. The n new vertices created will have k vertices which does not have out edges. Where k is the difference of in edges and out edges. The combing of the graph with vertices with the other case is carried out in the same way. Except in this case the preprocessor creates vertices which have no in edges. The figure below shows the working of the preprocessing function.

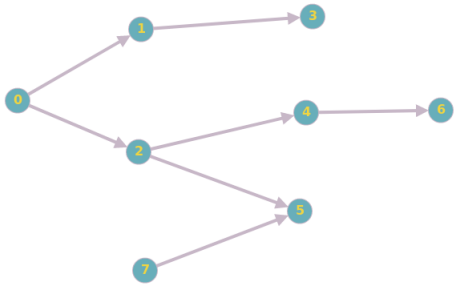

figure 6: The transformation of graph from figure 2 after applying the preprocessing

The node 2 in the example graph shown in figure 1 have 4 input edges and 2 output edges. The preprocessor creates two new vertices 11 and 12 which do not have any outgoing edges. The creation of new vertices is done assuming the combing width to be 2. The same process of the graph combing is carried out throughout all the vertices of the graph. The GPU accelerator is designed to overcome the bottlenecks of GPU. While processing the graph applications on GPU the load imbalance and too many kernel invocations are the main problems of graph computing. Let us see what are the graph processing bottlenecks and how the proposed graph accelerator overcomes the bottlenecks for computing.

The figure below gives the algorithm overview of the graph combing preprocessing algorithm. The Graph combing algorithm first computes the number of out going edges to each vertex of the input graph data set. Then the value of out going edges for each vertex is compared with the threshold. The threshold is desired number of outgoing edges per vertex we need to maintain or the threshold can be alternatively defined as the average number of outgoing edges per vertex after the combing operation is carried out. Now if the number of outgoing edges are greater than threshold then k or k+1 number of new nodes are created (i.e k in case of even number of outgoing edges, k+1 if odd number of out going edges). where k is the no of outgoing edges divided by the threshold. now each new node or the children node from the parent are allocated

to this newly formed children nodes. Alternatively this can be seen as the previous children of the parent node now actually become the children of the newly formed children nodes. After finished processing the current node the processing is moved to the next node in the order. This process is repeated for all the nodes in the graph.

Algorithm overview

thres=is the threshold number of edges

Each node can have.

For(loop iterates over number of nodes)

Starting from the first node

out_e=no of outgoing_edges(node)

Then,

k=out_e/thres

The out edges is split into k groups.

And k new nodes are created

And each new node is assigned with

The group elements

figure 7: The Graph combing methodology overview

The method described above works and can better the performance of the computations. but because of being naive it can support only few applications. Let us briefly look into the backdrops of the developed algorithm. The graph combing algorithm in a way to improve the performance by playing with the structure of the graph is limiting the applicability of graph combing to wide range of applications. the current developed method can only be applied to algorithms which do not depend on the structure of the graph.

For example the connected components problem will not depend on the structure of the graph rather depends only on existence or non existence of connections between the nodes. for applications which depend on the edge values we cannot obtain the same end result as the graph without any combing preprocessing applied. to encounter a part of this problem a improvised version of the graph combing algorithm is introduced. The naive implementation does not include the assigning and reading of edge values from the graph. So let us discuss in detail extra features of the new improvised graph combing algorithm.

The new version of graph combing algorithm is designed to make the graph combing algorithm applicable to graph algorithms which depend on the edge values for computing the end results. for example the Dijkstra's algorithm computes the shortest path from source vertex to all of the vertices in the graph. In this process of computing the shortest paths from source to all the vertices in the graph. The Dijkstra's tries to find the edge with minimum edge cost to add it to the path.

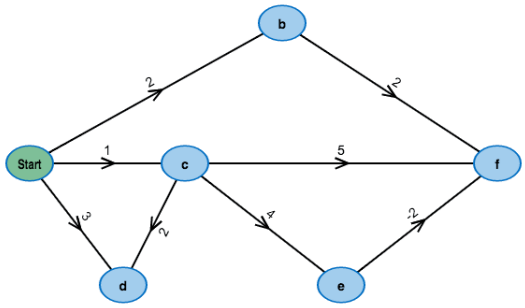

figure 8: Example to differentiate the naive and the improvised implementations of graph combing

For this let us consider the naive implementation. Let us suppose the naive implementation after creating the new nodes it assigns the edge values of zero to each and every new node created. Because of this the new nodes created all have a edge costs of zero. so the Dijkstra algorithm chooses one random edge out of the zero valued edges as its next path towards the goal. This can result into incorrect and unexpected results to many algorithms which have more likely the same working characteristic. To illustrate the working of naive graph combing, the naive graph combing transforms the node c of the above graph into the figure below graph combing.

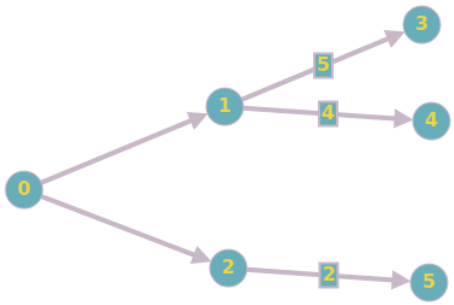

figure 9: The naive graph combing method of processing

As we can see from the above figure the graph combing naive implementation gives uncertain end results for Dijkstra algorithm and is the same case with many other such graph algorithms. So to counter this backdrop the graph combing algorithm in the improvised version instead of assigning the zero edge costs to all the edges pointing to the new nodes a distinct non zero value of negligible edge cost is added to each new node. To logically link the parent and the children nodes the new nodes which are holding the original children. we give edge values such that the edge value will be smallest for the new node which has the end node with minimum edge cost.

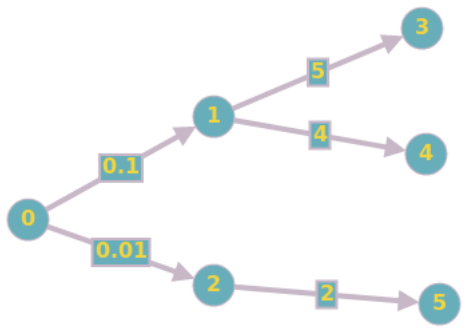

figure 10: The improvised Graph combing method of processing

Algorithm overview with edge values in mind:

thres=is the threshold number of edges

Each node can have.

For(loop iterates over number of nodes)

Starting from the first node

out_e=no of outgoing_edges(node)

Then,

k=out_e/thres

The out edges is split into k groups.

And k new nodes are created

figure 11: The improvised Graph combing methodology overview

As we can see the new way of combing the graph or modifying the structure of graph has actually able to increase the scope of applying the graph combing algorithm. The figure above shows the naive graph combing algorithm and the figure below shows the additional methodology introduced for the graph combing application to incorporate the graph combing for algorithms based on structures.

New nodes created are assigned edge values

greater than zero.

But this assignments does follow logic

Check for the new nodes created under a vertex

//first min

if(new node among the new nodes has the min edge

value node)

Edge value to new node=0.0001;

//second min

Edge value to new node=0.0002;

And so on for threshold number of nodes

figure 12: The improvised Graph combing methodology overview

4. Evaluation of the developed methodology:

The graph combing algorithm developed is actually tested on the connected components application. The connected components applications is tested on a data set of approximately 1005 nodes and 25000 edge vertices. initially the graph combing is carried out on the vertices of the graph as described above this creates a new edge list representation of the original data set. this new data set obtained is fed to the connected components algorithm to compute the number of connected components in the graph. The same is repeated with the original data set and the performance comparison is done between the two methods.

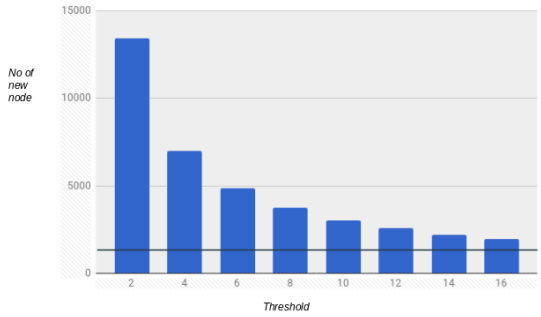

figure 13:connected components threshold vs number of edges

the above graph plotted tells us the relation ship between the number of nodes created as we go on vary the threshold value for the graph combing. so we can see when we try to have really small threshold values for the combing we end up creating two many

number of new nodes. This can result in poor performance due to the too much overhead created because of the new nodes creation. If we try to increase the threshold value the combing of the graph cannot bring in enough or desired parallelism into the application. so a balance between the threshold value should be obtained for achieving better performance results.

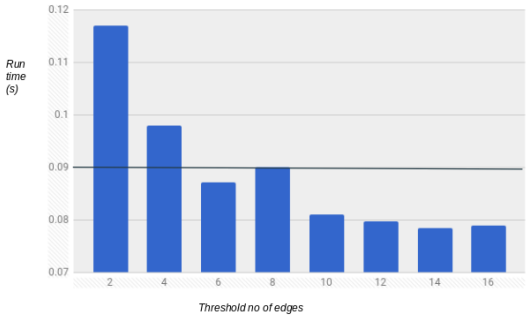

figure 14:connected components threshold vs runtime

in a similar fashion the plot of the threshold and the run time values for the connected components gives as another perspective that needs to be taken care while doing graph combing. Here we can see this as a consequence of the the previous. The increase in the threshold values is actually bringing the run time for computing the graph down. the red line in both the graph plots indicates the number of new nodes created and the run time of the uncombed graph data set. So the same can be inferred from the above plot. that is we need to be able to find a proper balance for the threshold value

5. CONCLUSION AND FUTURE NOTES

The graph combing method can be further extended by introducing new data structures to eliminate the short comings of the graph combing algorithm. Further the graph combing can be extended to graph processing on GPU. This methodology is an extension to the idea of Graphchi. The Graphchi paper methodology is developed in order to address the high latency of random reads and writes in accessing the graph data from disk. In this method we apply an extension of the idea to efficiently process the large Graph data on GPUs. When large graphs are processed on the GPU the main bottlenecks for the computing is caused due to the frequent invocation to kernel, improper utilization of SRAM i.e like registers, shared memory, constant memory, load imbalance within the warps and thread blocks. This method is aimed at overcoming the stated bottlenecks while computing Graphs on GPU.

The large graph which is stored in the disk is sampled into the GPU memory after going through preprocessing. The graph is initially divided into S approximately equal partitions and the first partition is brought in into the Global memory. Each interval s in the graph is associated with a shard. The shard stores the edges which have destinations in the interval in the order of the source. The remaining S-1 shards stores the output edges information of the first shard. As the shards stores the edge information in the order of the source. The access of the out edges of the sudgraph from the remaining shards will be sequential. This improves the spatial locality of the accessing the data from the disk. Once the sub graph is built in the global memory.

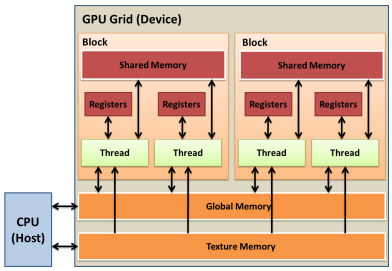

figure 15: The Global memory and the shared memory

The sub graph in the Global memory is further sub sampled into the L1 cache. By using the same methodology described above. In this way the subsampling can be carried out to all the levels in the memory hierarchy. But initially, the subsampling will be carried out up to two levels of memory hierarchy. At each level of subsampling the shard data from the previous subgraph is used to build the shard data for the new subsampled graph. Finally when the subsampled graph from the previous memory hierarchy is loaded into the shared memory. The structure of the graph is transformed by performing combing of the graph loaded into the shared memory.

The combing of the graph is carried out with a width of 1 edge. That is the sub graph is transformed into a graph which have vertices having only one in edge and out edge. However the vertices for which copies are made are entered into the table. The table stores the values of the new vertices created and the parent vertex. When the execution of the sub sample in the shared memory is complete updating the subsample graph is carried out using the table. Here the sampling is done based on the size of the shared memory.

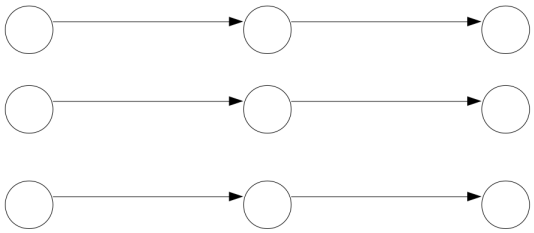

figure 16: The graph after applying the pre processing looks similar to the above figure

By changing the structure of the graph we are increasing the parallelism in processing each sub sample of the graph. As the shared memory is the memory available to the threads in the same thread block. Through this method the theoretical expectation is by structural transformation of the sub graph in the shared memory we can reduce the load imbalance within a thread block and increases the SIMT line utilization.