Exploring Hybrid Hardware and Data Placement Strategies for the Graph 500 Challenge

Scott Sallinen scotts@ece.ubc.ca
Daniel Borges dlsbdaniel@gmail.com
Abdullah Gharaibeh abdullah@ece.ubc.ca
Matei Ripeanu matei@ece.ubc.ca

I. Introduction

This paper presents our experience with exploring the configuration space and data placement strategies for Breadth First Search (BFS) on large-scale graphs in the context of a hybrid, GPU+CPU architecture and the Graph 500 challenge [1]. We show that performance depends heavily on multiple, non-intuitively interrelating factors: the partitioning strategy used, the graph's layout on memory, and on the type of memory used to allocate the GPU partition.

Recent work on GPU graph traversal has often targeted relatively small graphs that fit on device memory only. Additionally, some of this past work (e.g., CUSHA [6]) uses space intensive data representations that trade a larger memory footprint for higher performance, an impractical approach for the Graph500 challenge. Our goal is to process large graphs that stretch the limits of single-node commodity machines.

When processing large graphs with tens of billions of edges or more, the techniques we explore are crucial to obtain an optimal performance level on hybrid CPU+GPU architectures. In particular, we show that the following configuration options can significantly impact performance: (*i*) the partitioning strategy, for up to a 1.7x improvement, (*ii*) the placement of the graph partitions in memory in a 'sorted by degree' order, for up to 3.5x improvement, and (*iii*) the strategy to manage memory used for the GPU partitions when the graph no longer fits in the device memory, for up to another 1.6x improvement.

II. Experimental Platform and Workload

Software platform. We use TOTEM [4], a generic graph processing framework that simplifies the task of implementing graph algorithms for single-node platforms that include processing elements with separate memory spaces (e.g. discrete GPUs). TOTEM uses the Bulk Synchronous Parallel (BSP) model and provides key functionalities to implementers: a common data representation, ghost nodes and their synchronization, hooks to callbacks that implement algorithm-specific CPU and GPU kernels, and optimizations common across algorithms (e.g. message aggregation, and overlapping computation with communication).

BFS implementation. The BFS algorithm we use is standard, level-synchronized, and classic in nature. The CPU and GPU each have their own, fairly optimized, kernel. They are based on the implementations proposed by Hong et al. [7, 8]. TOTEM is open source and these kernels can be viewed online [2].

We note that, since the BFS algorithm we chose is generic and based on BSP, the performance we obtain is unlikely to beat the performance of specialized BFS kernels (e.g., GRAPHCREST [3], which implements a Graph500 specific kernel). It should be noted that this work focuses on exploring optimized data partitioning and placement techniques using TOTEM's generic model, and in future work we will optimize the algorithm itself.

Hardware platform. We use a machine with two Intel Xeon E5-2670 processors (2.5 GHz, 10 cores), and two Nvidia Kepler K40's (745 MHz, 2880 cores). The host has 512GB of memory while the GPUs have 12GB each.

Workload. We generate graphs as for Graph500 (i.e., similar parameters for the Kronecker graph generator). Graphs are power-law with an average node degree of 16. Here we focus on graphs of Scale 30: this means 1 billion vertices and 16 billion undirected edges. Using the memory efficient Compressed Sparse Row (CSR) graph representation (Fig. 2), these graphs are approximately 256GB; thus we cannot fit the entire graph or even a significant portion into a single GPU. While we present performance only for Scale 30, our experiments show that the trends we present hold for at least one order of magnitude: from Scale 27 to 30.

Performance metric. Similar to the Graph500 benchmark, we report the raw processing rate measured in Traversed Edges Per Second (TEPS) with the only difference being that we count directed edges instead of undirected ones.

III. Exploring the Configuration Space

The preprocessing pipeline for TOTEM has two stages: loading the graph in memory in the CSR format, and then partitioning it for the hybrid platform. During preprocessing the degree of each vertex is calculated, and used to compute a prefix sum to determine the relevant edges for a given vertex. If specified, in this phase vertices are also sorted by degree during the memory transition. To represent undirected edges, each edge is listed twice, once in each direction.

A. How much and which part of the graph to offload to GPU?

Fig. 1 shows that simply distributing the vertices naïvely at random across the processing elements, significantly hampers performance. The alternative is to allocate high or low degree vertices to the CPU. For our large graphs, the best choice is to have the CPU focus on the low degree vertices, and thus present the higher degree vertices to the GPU. This offers a 1.7x speedup over the naïve approach. This is in contrast to our past work on smaller (Scale27) graphs [5] which finds a cache improvement with placing the high degree nodes on the CPU. However, for large graphs the cache effect diminishes. As we begin to use mapped memory (§III.C), we achieve higher performance by taking advantage of the massive-multithreading of GPUs for processing the high degree vertices by processing their many neighbors in parallel.

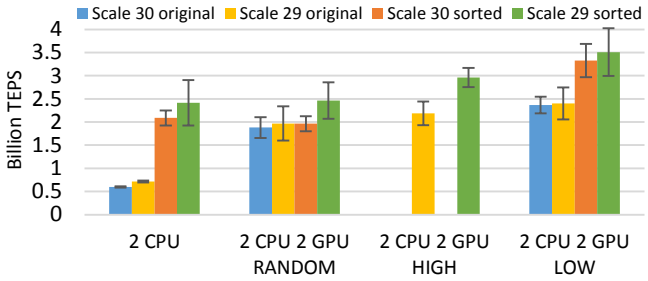

Fig. 1: The effect of partitioning and sorting for graphs of scale 29 and 30. The original bars show average performance when vertices are not sorted. The bottom labels represent the partitioning technique: random, high, or low degree vertices on CPU. High partitioning requires more than available GPU memory for scale 30, therefore the bars are missing. The error bars show STDDEV over 10 experiments.

B. What is the Effect of Memory Layout on Performance?

The previous section shows that, for a hybrid configuration, informing graph partitioning by vertex degree brings performance gains. When building partitions we can also sort vertices by degree, and remap the memory layout of the graph with negligible time overhead and $O(v)$ additional memory (Fig. 2). During the post processing phase, we can remap the results back to their original IDs, as the strategy only changes the memory layout and not the graph itself.

This sorted layout offers a massive performance boost for graph traversal, showcasing an improvement of up to 3.5x, as show in Fig. 1 and Fig. 3. The hybrid configurations receive a lower, yet still significant boost of 1.3x on average.

Our initial hypothesis on the reason behind the significant performance boost the CPU-only configuration achieves compared to the hybrid one after sorting is that by partitioning the graph across the CPU and GPUs, the hybrid configuration already creates high level split of vertices by degree, which in practice is equivalent to an approximate sort, and hence the hybrid configuration does not benefit much from sorting compared to the CPU-only one. However, by using a most-significant-bit first radix sort with a variable precision, we found that approximate sorting using the first half of bits showcased less than 1.1x improvement. In fact, the results only approached the fully sorted improvement near 29 out of 32 bits. We suspect this is due to locality and caching, and we plan to explore this necessity of high precision sorting in future work (e.g., using performance counters).

C. How can we utilize mapped memory?

Fig. 3 presents the performance for different configurations that use CUDA's memory mapping technique. This allows a GPU to transparently access the host DRAM over the PCIe bus (i.e., without explicit copies). As a result, this option offers new configuration options: what part of the graph representation (vertex array, edge array) to store in the device memory, and what part to memory-map.

In effect, device memory (i.e., 'global' GPU memory) is used as another level of software managed cache: frequently accessed information is placed on the device memory and the rest is retrieved from the host only when necessary. Fig. 3 shows that if we can at least store the vertex array on the GPU's (with the edge array memory mapped), we can achieve

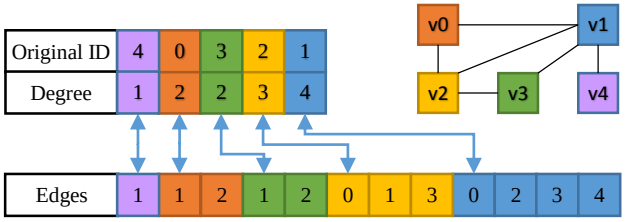

Fig. 2: A sorted Compressed Sparse Row format, ordered by degree (low to high) instead of by vertex ID.

a boost in performance of 1.5x over a CPU only version, and additionally placing as many relevant edges as possible in the remaining memory of the device, gives a 1.6x improvement.

IV. CONCLUSION

We show the significant performance impact of the different configuration options (i.e., vertex sorting, graph partitioning strategy, use of memory mapping and memory placement) for efficient large scale graph processing on hybrid platforms. We showcase multiple compounding performance gains that each range from 1.3x to 3.5x depending on the baseline and the configuration knob used.

It is also notable that these techniques are applicable for more than just Breadth First Search or the Graph500 challenge. Considering memory placement for other graph algorithms is important with larger scales, and considering which hardware is responsible for certain vertices can be dependent on the type of algorithm in use.

V. REFERENCES

[1] http://www.graph500.org

[2] http://netsyslab.ece.ubc.ca/wiki/index.php/Totem

[3] http://www.graphcrest.jp/eng/index.html

[4] A. Gharaibeh, et al. *A yoke of oxen and a thousand chickens for heavy lifting graph processing.* (PACT 2012).

[5] A. Gharaibeh, et al. *On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest*. (IPDPS 2013).

[6] F. Khorasani, et al. *CuSha: vertex-centric graph processing on GPUs.* (HPDC 2014).

[7] S. Hong, et al. *Accelerating CUDA graph algorithms at maximum warp*. (PPoPP 2011).

[8] S. Hong, et al. *Efficient Parallel Graph Exploration on Multi-Core CPU and GPU.* (PACT 2011).

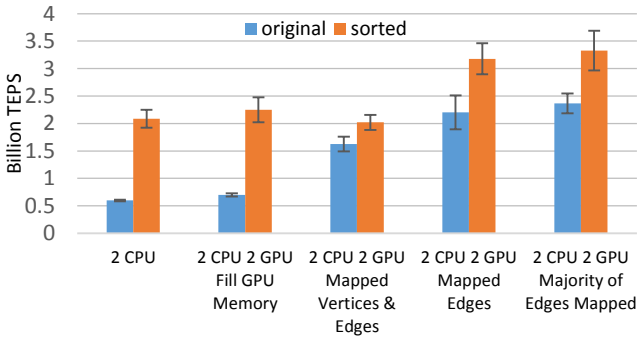

Fig. 3: The impact of memory mapping and sorting on performance for a scale 30 graph versus device memory. The original bars show average performance when vertices are not sorted. The bottom labels represent the memory mapping technique: all data is memory mapped, or allocating the vertex array and (fully or partially) mapping the edge array to the host. The error bars show STDDEV over 10 experiments.